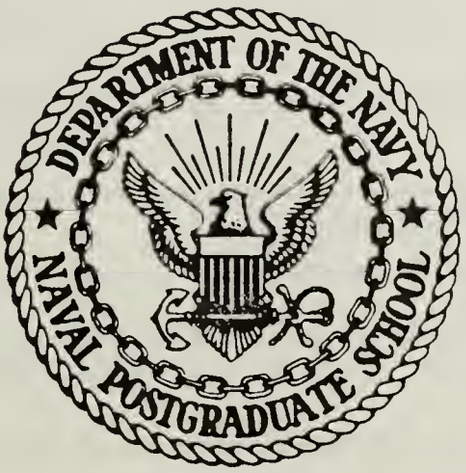
1986

# Computer graphics interactive workshop for two dimensional fractals.

Mason, Lewis Gerhard.

# NAVAL POSTGRADUATE SCHOOL
## Monterey, California




# THESIS

COMPUTER GRAPHICS INTERACTIVE WORKSHOP
FOR TWO DIMENSIONAL FRACTALS

by

Lewis Gerhard Mason

December 1986

Thesis Advisor: Michael J. Zyda

unclassified
ECURITY CLASSIFICATION OF THIS PAGE

## REPORT DOCUMENTATION PAGE

| | |
|---|---|
| a REPORT SECURITY CLASSIFICATION: unclassified | 1b RESTRICTIVE MARKINGS: |
| a SECURITY CLASSIFICATION AUTHORITY: | 3 DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution is unlimited. |
| b DECLASSIFICATION/DOWNGRADING SCHEDULE: | |
| PERFORMING ORGANIZATION REPORT NUMBER(S): | 5 MONITORING ORGANIZATION REPORT NUMBER(S): |
| a NAME OF PERFORMING ORGANIZATION: aval Postgraduate School; 6b OFFICE SYMBOL (If applicable): 52 | 7a NAME OF MONITORING ORGANIZATION: Naval Postgraduate School |
| : ADDRESS (City, State, and ZIP Code): Monterey, California 93943-5000 | 7b ADDRESS (City, State, and ZIP Code): Monterey, California 93943-5000 |
| NAME OF FUNDING/SPONSORING ORGANIZATION; 8b OFFICE SYMBOL (If applicable): | 9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: |
| ADDRESS (City, State, and ZIP Code): | 10 SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO; PROJECT NO; TASK NO; WORK UNIT ACCESSION NO |

TITLE (Include Security Classification): COMPUTER GRAPHICS INTERACTIVE WORKSHOP FOR TWO DIMENSIONAL FRACTALS

PERSONAL AUTHOR(S): Mason, Lewis Gerhard

| a TYPE OF REPORT | 13b TIME COVERED FROM ____ TO ____ | 14 DATE OF REPORT (Year, Month, Day) | 15 PAGE COUNT |
|---|---|---|---|
| laster's Thesis | | 1986 December | 112 |

SUPPLEMENTARY NOTATION:

| COSATI CODES | | |
|---|---|---|
| FIELD | GROUP | SUB-GROUP |
| | | |
| | | |

18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number): fractals; fractal dimension; recursion; Mandelbrot set; Julia sets


ABSTRACT (Continue on reverse if necessary and identify by block number)

This study presents a background for fractals and describes an interactive computer graphics workshop for two-dimensional fractals. The workshop enables the user to learn about fractals through experimentation with the generation of Koch-like fractal curves. A variety of Koch-like fractal curves, Julia sets and the Mandelbrot set are presented as examples. Algorithms are presented for creating the Mandelbrot set and for creating Koch-like fractal curves.


| | |
|---|---|
| D STRIBUTION/AVAILABILITY OF ABSTRACT: ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS | 21 ABSTRACT SECURITY CLASSIFICATION: unclassified |
| NAME OF RESPONSIBLE INDIVIDUAL: Michael J. Zyda | 22b TELEPHONE (Include Area Code): (408) 646-2174; 22c OFFICE SYMBOL: Code 52Zk |

FORM 1473, 84 MAR — 83 APR edition may be used until exhausted; All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE: unclassified

# Computer Graphics Interactive Workshop For Two Dimensional Fractals

by

**Lewis Gerhard Mason**
Commander, United States Navy
B. S.,United States Naval Academy, 1971

Submitted in partial fulfillment of the
requirements for the degree of

**MASTER OF SCIENCE IN COMPUTER SCIENCE**

from the

NAVAL POSTGRADUATE SCHOOL

December 1986

## ABSTRACT

This study presents a background for fractals and describes an interactive computer graphics workshop for two-dimensional fractals. The workshop enables the user to learn about fractals through experimentation with the generation of Koch-like fractal curves. A variety of Koch-like fractal curves, Julia sets and the Mandelbrot set are presented as examples. Algorithms are presented for creating the Mandelbrot set and for creating Koch-like fractal curves.

## TABLE OF CONTENTS

# I. INTRODUCTION

## A. WHY DO WE NEED FRACTALS?

If you have ever tried to draw a picture of a natural object on a computer graphics terminal with only the primitive drawing commands normally found on a computer, then you know how hard it is to create a realistic looking picture of a natural object. Why is this so difficult? Benoit B. Mandelbrot, who in 1975 coined the term ***fractals*** to give a title to his first essay on the subject says, "Clouds are not spheres, mountains are not cones, coastlines are not circles, and bark is not smooth, nor does lightning travel in a straight line" [Ref. 1: p. 6]. In general, the objects of nature are not created with the regularity of the Euclidean primitives.

Clearly then something new is needed if we are going to draw natural objects and make them look more realistic. The standard Euclidean geometry with its primitives, lines and circles and cones, and three dimensions is not enough for either describing or rendering realistic looking objects.

## B. WHAT ARE FRACTALS?

### 1. What are their Origins?

Mandelbrot is a mathematician, who has drawn heavily on his mathematical background to describe fractals. He has relied on the works from

the mathematicians of the late 1800's and early 1900's. This was a period when several maverick mathematicians challenged the conventional ideas of their day. These mavericks, whose works Mandelbrot uses, Cantor, Peano, Weierstrauss, Julia, Fatau and von Koch all proposed ideas that went against the establishment at the time. They invented ideas such as continuous curves that had no derivatives, and lines that could fill whole areas of a plane. These functions were looked on as a "gallery of monsters", "pathological". "psychotic" and even "terrifying". However, in about 1925 their ideas were placed on the shelf and largely forgotten. In 1975, Mandelbrot began studying these old "monsters". He had computers to use as a tool for the multitude of calculations these functions required. He also used the precision of computer graphics to display the results. The results give a visual, intuitive feel to these "monsters". Illuminated by computer graphics, they have finally been recognized as some of the basic structures in the language of nature's irregular shapes, the "Fractal Geometry of Nature" [Ref. 2: p. 806].

2. The Meaning of Fractal

Mandelbrot said he coined the term fractal to represent something "rough but self-similar". He also says that both terms are needed to describe fractals. Intuitively, roughness is a measure of what an object looks like. Self-similarity, simply put, is "as you zoom in and examine any portion, it doesn't look different".

Roughness can best be explained with a few examples. A straight line is a standard Euclidean object in one dimension and is not considered rough. A plane is a standard two dimensional object and is not considered rough. What about a line that wanders around the plane? It is no longer a simple line, yet it does not fill the plane either. Roughness is a representation of the "wiggliness" of the line and is related to a fractal dimension. Fractal dimension is formally defined in [Ref 1: p. 361], but is considered here informally as a measure of roughness. A relatively straight line. with few "wiggles" has a roughness greater than the straight line, and has a fractal dimension greater than 1.0. For example, Figure 1.1 is a relatively straight line and has a fractal dimension of approximately 1.06. A line that has many "wiggles" has a roughness that is greater than a straight line and also greater than a line with less "wiggles". Figure 1.2 is an example of a line "more wiggly" than Figure 1.1. The line in Figure 1.2 has a dimension of about 1.26. As the roughness of a line gets larger, the line wiggles all around the plane and almost completely fills a portion of the plane, looking much like a filled polygon in Euclidean primitives. A line with a lot of roughness approaches 2.0 for a fractal dimension as it approaches filling a portion of the plane. The same concept of roughness or dimension between 1.0 and 2.0 can be applied to change in roughness from a plane (2.0) to a solid (3.0). A plane that is "crumpled" is no longer a smooth plane, and has a roughness greater than two but less than three. In this study, we concentrate on lines with dimensions between 1.0 and 2.0.

Figure 1.1 Dimension = 1.06

Figure 1.2 Dimension = 1.26

There are two types of self-similarity: exact self-similarity and statistical self-similarity. Exact self-similarity for a Fractal curve is achieved when the rules of construction make the curve out of parts that look just like, but reduced in size from, the original. A good example of exact self-similarity is the von Koch snowflake (Figure 1.3). The von Koch snowflake was first proposed in 1904. To describe the snowflake, some of Mandelbrot's terminology needs to be introduced. There are two main components used in the construction of the snowflake. The "initiator" is the segment with which you start. The "generator" is the object that is substituted in for the initiator, properly scaled so the end points of the initiator and generator are coincident. An ititial object is composed of one or more initiators. The initial object for the snowflake is a triangle, and the initiator for the snowflake is a straight line segment. The generator is four smaller line segments each scaled to $\frac{1}{3}$ of the length of the original line segment and connected as depicted in Figure 1.3. The snowflake is created by starting with an equilateral triangle. Each of the sides of the triangle is considered an initiator. A generator is substituted in for each initiator. The object then has 12 line segments each being $\frac{1}{3}$ of the original length. Each of the 12 line segments is now considered an initiator and the replacement is repeated. In theory, the replacements can be repeated an infinite number of times, since for any two real numbers there are infinitely many real numbers between them. This produces a curve that has an exact self-similarity. If any portion of the curve is expanded to

Figure 1.3 The Koch Snowflake

an arbitrary magnification it looks exactly the same as the original curve. Most objects in nature do not have this exact self-similarity, they exhibit a statistical self-similarity.

A good example of statistical self-similarity is a tree. A branch of a tree with its smaller branches looks a lot like the trunk of the tree with its branches. Another example that is used in the literature is a coastline. A small segment of a bay, when examined in detail looks similar to a large section of a coastline in a larger scale. Although the segments being viewed are not exactly alike, they do look remarkably similar. The key point is that fractals exhibit an invariance under changes of scale.

## C. ARE FRACTALS BEING USED NOW?

Fractals are being used now. Within the last five to ten years fractal geometry and its concepts have become central issues in most of the natural sciences. Fractals can "mimic the activities of the stock market, the motion of molecules, and the growth of plants. Consequently their use ranges from physics, biology, and sociology, to art and even motion-picture scene simulation" [Ref. 3: p. 157]. Fractals have proven useful in two ways, both describing and acting as a mathematical model for many seemingly complex shapes found in nature.

## D. GOALS OF THIS RESEARCH

Using fractals would be almost impossible without the aid of computers. While the formulas creating fractal curves are simple, the calculations must be performed over and over, each time using the result of the previous calculation. All but the simplest are calculated millions of times. Once the calculations are made, the rendering of the pictures requires the precision of computer graphics for proper execution.

Gaddis states:

> There are two approaches that can be taken in the investigation of fractal geometry and computer graphics.
> - To view the computer as a tool to enhance the investigation of fractal geometry.
>
> or
>
> - To view fractals geometry as a tool to enhance the realism of computer graphics. [Ref. 4: p. 6]

In this study, we have chosen to follow the first approach, to use the computer to aid in learning about fractal geometry through experimentation with an interactive workshop on two dimensional fractals.

We have created an interactive workshop where an individual can sit at a computer terminal and using a mouse for inputs, sketch a generator directly onto the video screen. If the generator is satisfactory, the initiator structure can then be selected and the replacements commenced. The replacements can not continue indefinitely, and we use a termination condition such that when the length of the new initiator is less than one pixel width on the screen the replacement is stopped.

The results can be rapidly observed and the experiment can be repeated with a new generator if desired. There is also a facility to save the results to a file in a format that our laser printer can take as input. Hard-copy results of the experiments can be saved and compared.

## II. ABOUT KOCH-LIKE FRACTALS

### A. FRACTAL DIMENSION

The following section is not a rigorous discussion of fractal dimension. It is aimed at giving an intuitive feeling of a fractal dimension and self-similarity. Self-similarity is the property that a portion of an object examined under an arbitrary magnification looks similar to the original. The example cited most often in the literature is the coastline of Great Britain. Looking at a map that shows the entire west coast of Great Britain, and comparing it to a map that shows only the south one-quarter of the coast expanded to occupy the same area on a page, the coastlines look vaguely similar. They have the same "ruggedness". Continuing on in the same manner and examining an arbitrary bay, expanding the view of the bay shoreline, a similarity, the same "ruggedness" is noticed. Self-similarity is observed independent of the viewing scale. Fractal curves with exact self-similarity are invariant under changes of scale. Fractal curves that are not exactly self-similar are not invariant, but they still resemble each other under changes of scale.

Scaling also affects the length of the perimeter of the island. A drive around the island is shorter than a walk that closely follows the shoreline. The perimeter determined by the walk is shorter than the perimeter determined with a set of

calipers measuring around the island, along the waterline. The smaller the measuring scale, the longer the perimeter appears. This concept of decreasing scale and increasing perimeter can be transported to fractal curves. The scale factor is closely related to the fractal dimension. For example, in one dimensional space, start with a finite line segment. Divide it into N equal parts. Each part is a smaller version of the original line, scaled down by a ratio $r = 1/N$. A general formula in one dimension is $N\ r^1 = 1$. (Figure 2.1).

Using the same idea on a two dimensional object yields similar results. Begin with a two dimensional square. Divide it into N equal parts. (For example divide it into 4 equal parts by connecting the midpoints of each line segment.) Each part is a smaller version of the original square, scaled down by a ratio $r = \frac{1}{N^{\frac{1}{2}}}$ (for our example $N = 4$, so $r = \frac{1}{4^{\frac{1}{2}}}$ or $r = \frac{1}{2}$). For two dimensions the formula is $N\ r^2 = 1$. (Figure 2.2).

Going one step more into the third dimension follows the pattern already established. Begin with a cube, all sides of equal length. Divide the cube into N equal parts. (For example construct 8 equal parts by connecting the midpoints of each side.) Each part is a smaller version of the original cube, scaled down by a scaling ration $r = \frac{1}{N^{1/3}}$. (For our example $N = 8$, $r = \frac{1}{8^{1/3}}$ or $r = \frac{1}{2}$.) Again for three dimensions $N\ r^3 = 1$. (Figure 2.3).

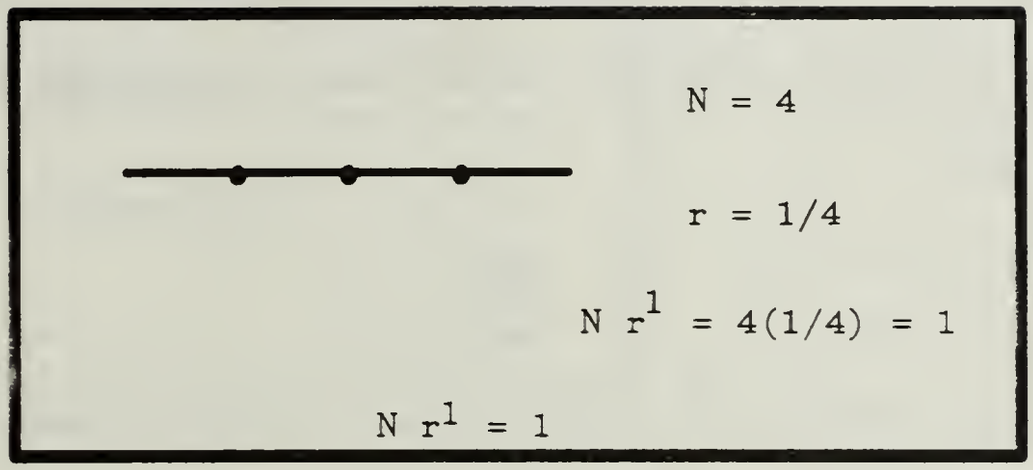



Figure 2.1 One Dimension

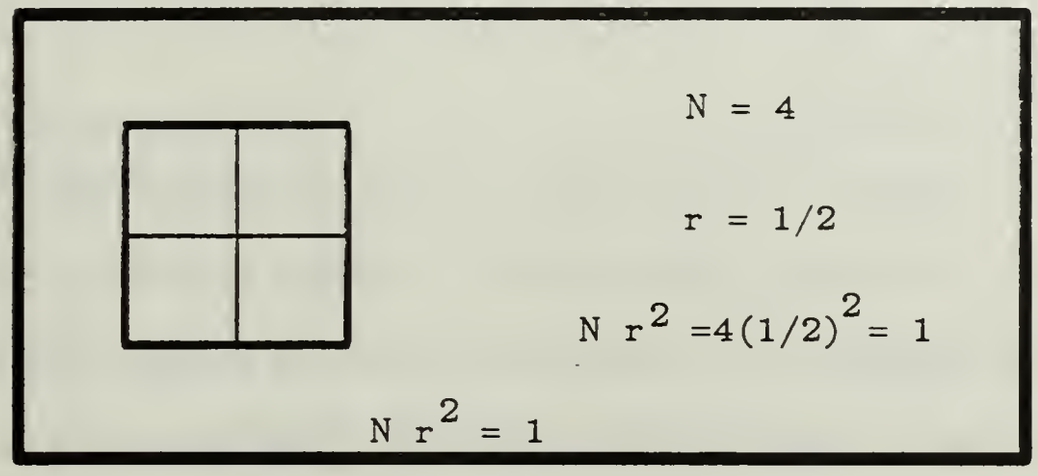



Figure 2.2 Two Dimensions

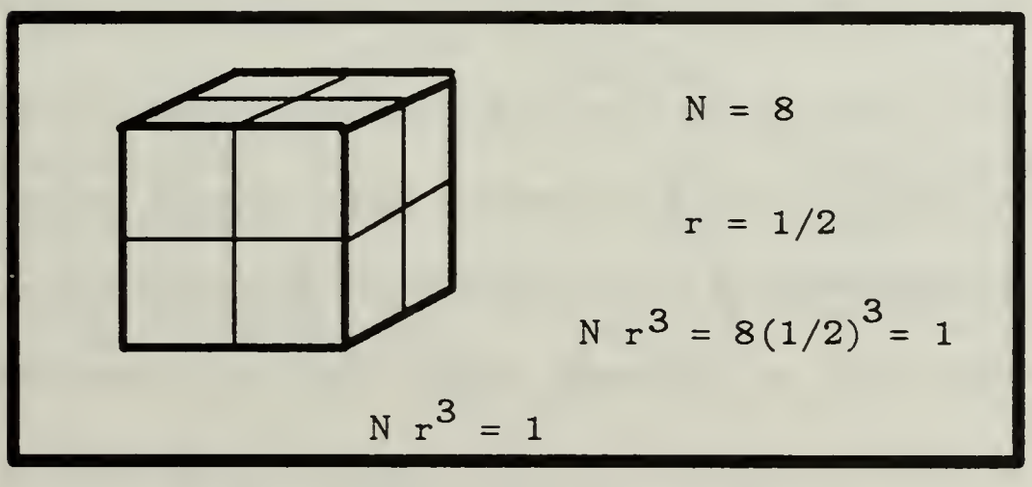



Figure 2.3 Three Dimensions

In general if N is the number of equal parts, and r is the scaling ratio, we can let D represent the dimension and the equation becomes $N\,r^D = 1$. If we now assume that dimension is not limited to integer values we can solve the equation for D. The result is

$$D = \frac{\log N}{\log(\frac{1}{r})}$$

This D, that depends on the scaling ratio and the number of segments, is a good measure of the fractal dimension.

## B. THE KOCH CURVE

We now return to the Koch curve for some examples of the dimension calculations. The Koch curve is created by a recursive replacement technique. Again we use Mandelbrot's two components, initiator and generator. The initiator in our examples is always a single line segment. The generator is four line segments. Each of those four line segments is $\frac{1}{3}$ the length of the initiator. For all the Koch-like curves, the end points of the initiator become the end points for the generator. That is, a single line segment (the initiator) is replaced by an appropriately scaled down set of line segments (the generator). The von Koch snowflake begins with an equilateral triangle. Each leg of the triangle is considered an initiator, thus there are three initiators at the begining of the construction of the curve (Figure 2.4) [Ref. 4: p. 26].

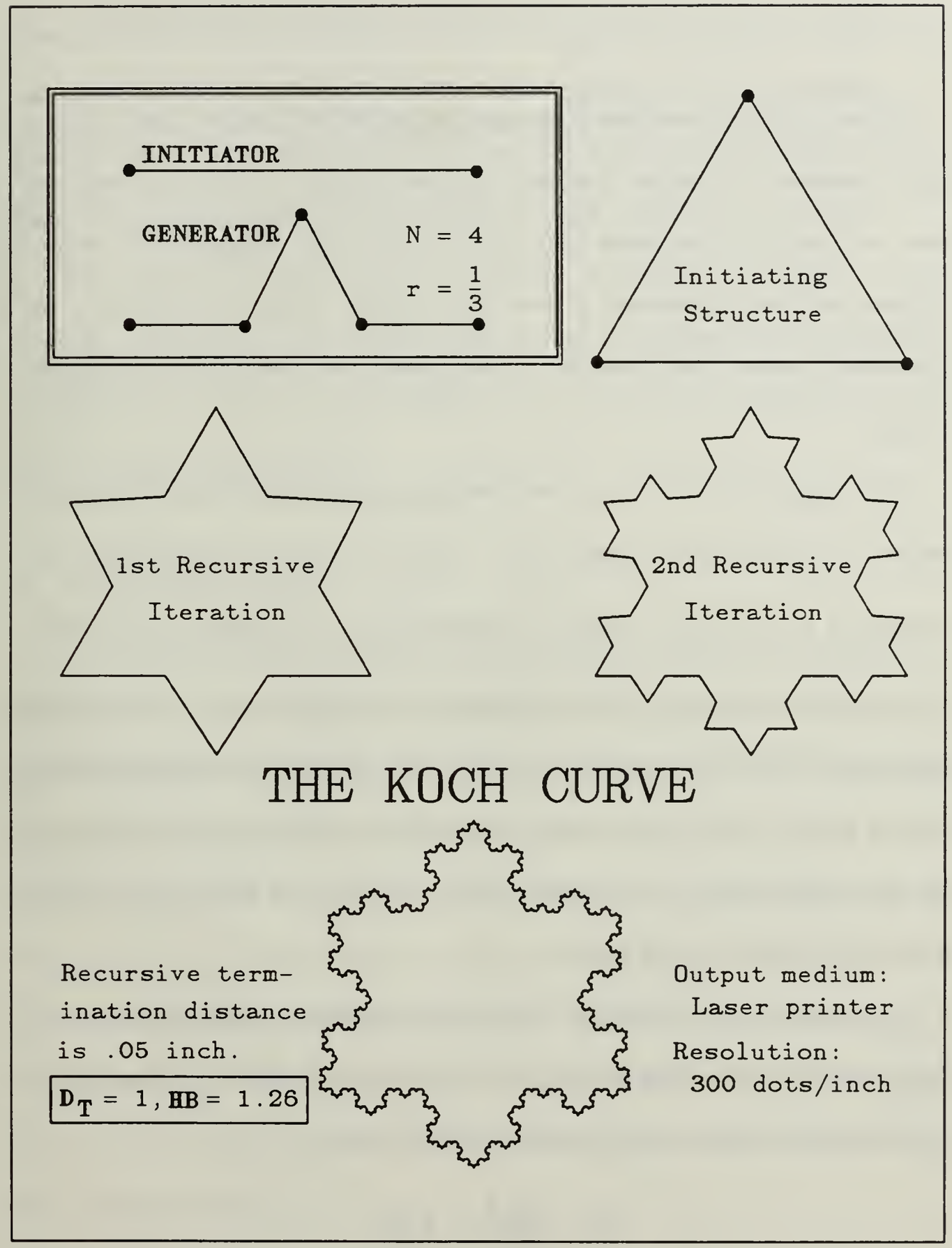



Figure 2.4 The Koch Curve.

We begin with the equilateral triangle and replace each of the three initiators with a generator. The picture now looks like the perimeter of a star of David, and there are now 12 line segments (Figure 2.4). The 12 line segments now become initiators and the replacement is done again. Each initiator is replaced by an appropriately scaled down generator. This process is repeated an infinite number of times.

If we assign the value of **1** to each beginning line segment's length, then the perimeter of the original triangle is **3**. After the first set of replacements the perimeter is **4**. After the second set of replacements the perimeter is $5\frac{1}{3}$. With each successive iteration of the replacements, the length of an individual line segment decreases (and approaches 0) while the length of the perimeter grows without bounds. This is the seeming contradiction that the mathematicians of the early 1900's found so interesting. The results do not fall nicely into the 'normal' three dimensions of Euclid.

If we examine the dimension of the initiator generator relationship, we see some possible answers. The number of line segments is 4, and the scaling ratio is 1/3. Using these values in the dimension equation gives

$$\mathbf{D} = \frac{\log \mathbf{4}}{\log \mathbf{3}} \approx \mathbf{1.26}$$

This is certainly not an integer but then the curve does not behave in Euclidean fashion. Figure 2.5 is an example of the representation of a Koch snowflake on a graphics screen. The resulting picture is certainly not a line, yet it does not fill a plane so it is somewhere between a dimension of 1.0 and a dimension of 2.0.

The 1.26 dimension represents a "wiggliness" of the line in the plane. As the number grows from 1.0. the curve begins to "wiggle" through more of the plane. A dimension of 1.26 is not very "wiggly" but a curve with a dimension near 2.0 almost becomes a solid polygon.

## C. KOCH-LIKE CURVES

All Koch-like fractal curves use the same recursive replacement techniques as described above. The dimension calculation for a Koch-like curve that does not have all segments of the generator the same size is not as nice. Since the segments of the generator are not the same size, the scaling ratio is not a fixed number. There have been several methods proposed for determining the fractal dimension when the scaling ratio is not constant throughout the generator. One such method uses the relationship between the perimeter length and the size of the measuring tool. "When using a ruler of size **r** to measure a coastline's length, the total length equals the ruler size times the number of steps of size **r**, **N(r)**, taken tracing the coast.

$$\mathbf{LENGTH} = \mathbf{r}\ \mathbf{N(r)}.$$

Figure 2.5 The Koch Snowflake

As with the snowflake, $N(r)$ varies on the average as $\frac{1}{r^D}$ and

$\mathbf{LENGTH} = r\,\frac{1}{r^D} = \frac{1}{r^{D-1}}$." [Ref. 2: p. 808] The approximate fractal dimension can be determined from this equation.

## III. JULIA SETS AND THE MANDELBROT SET

### A. HISTORY

The Koch-like curves are not the only type of fractal. There are two types of fractals that have been used to produce the majority of the fractal pictures seen in the literature. Fractal mountains are generated through the use of "random" fractals. Random fractals are the ones that imitate nature [Ref. 3: p. 157]. The technique for producing random fractals is described in [Ref. 4]. The second type of fractal produces the dragons and multicolored monsters of the Julia sets and the Mandelbrot set. Julia sets and the Mandelbrot sets are fractals that are invariant under non-linear transformations. Mandelbrot has used the recursive, non-linear relation $Z_{n+1} = Z_n^2 + C$ , where Z and C are elements of the complex number system. This function was first studied by G. Fatau in 1906 [Ref. 2: p. 224]. The sets formed by this function were studied in "the theory of iteration of rational maps of the complex plane". The study reached its high point in 1918, with papers by both G. Julia and P. Fatau [Ref. 5: p. 153]. These sets have become known as Julia sets. When Mandelbrot first started creating Julia set pictures he says:

> We accumulated beautiful drawings of Julia sets by the bushel. . . . It was nice to understand intuitively, at long last, what Julia and Fatau had really been after. And in addition nearly all Julia sets proved to be extraordinarily beautiful. [Ref. 5: p. 153]

To create these beautiful pictures, we need to understand a little about the complex numbers and the complex plane.

## B. THE COMPLEX PLANE

The complex plane is a two dimensional plane with the two axes labeled real and imaginary. All complex numbers can be represented as a point in the complex plane and are of the form $Z = a + bi$ . $a$ is the component along the real axis and $b$ is the component along the imaginary axis. The $i$ is a representation of $\sqrt{-1}$. such that $i^2 = -1$. Addition and subtraction in the complex plane is similar to addition in the "Real" number system. except real components are added to real components and imaginary components are added to imaginary components. For example, if $Z_1 = a + bi$ and $Z_2 = g + hi$ then $Z_1 + Z_2 = (a + g) + (b + h)i$. Multiplication is more complicated and is carried out as the multiplication of two ordinary binomials. For example, again using $Z_1$ and $Z_2$,

$$Z_1 \times Z_2 = a(g + h)i + b(g + hi) = ag + ahi + bgi + bhi^2.$$

Since $i^2 = -1$,

$$Z_1 \times Z_2 = (ag - bh) + (ah + bg)i.$$

The modulus of a complex number is a measure of the distance of the point representing that number from the origin of the plane. The modulus can be calculated by using the Pythagorean theorem,

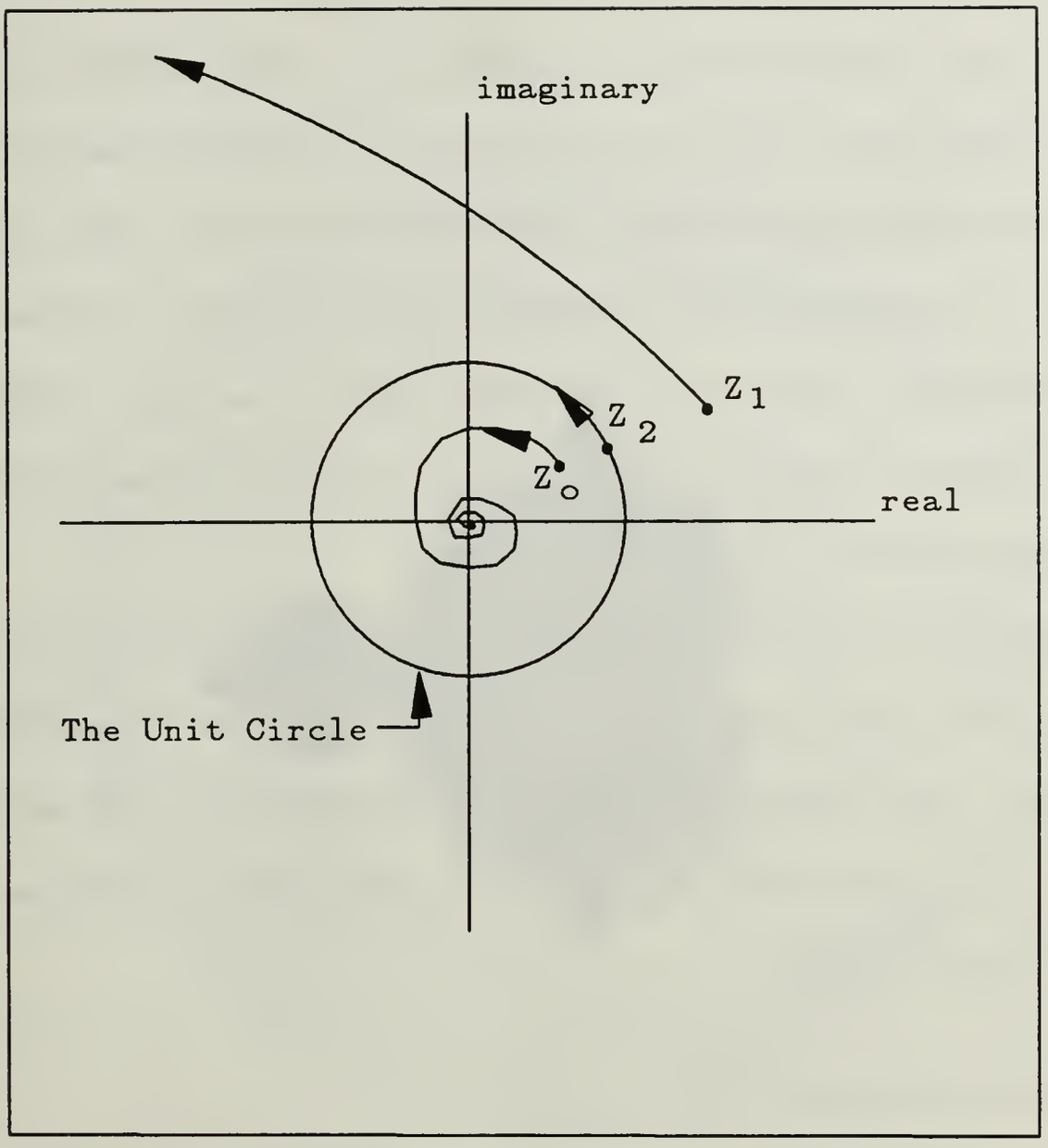
$$\textbf{modulus } |Z_n| = \sqrt{a^2 + b^2}.$$

C. JULIA SETS

What is a Julia set? The easiest answer is a simple example. For the function $Z_{n+1} = Z_n^2 + C$ pick the value for C to be $0 + 0i$. The function then becomes $Z_{n+1} = Z_n^2$ . a simple squaring of the complex number. Consider the starting value, $Z_0$ in polar coordinates. a modulus and an angle measured from the real axis, representing the point. To obtain the value $Z_0^2$ we simply square the modulus and double the angle. If the function is iterated time and time again there are three possible outcomes.

1. If the modulus of $Z_0$ is greater than 1, then $Z_{n+1}$ approaches $\infty$.
2. If the modulus of $Z_0$ is less than 1, then $Z_{n+1}$ approaches 0.
3. If the modulus of $Z_0$ is equal to 1, then $Z_{n+1}$ is on the unit circle about the origin and remains on the circle forever. (Figure 3.1)

In this example, the two values 0 and $\infty$ act as attractors if Z is not on the unit circle. With multiple iterations of the function, Z begins to approach either 0 or $\infty$. 0 and $\infty$ attract values, and they each have a basin of attraction. The basin of attraction of a limit value is the set of all numbers that when iterated repetitively approach that limit value. In our example, everything inside the unit circle is in the basin of attraction of 0. Similarly everything outside the unit circle




$Z_o$ after multiple iterations approaches Zero.

$Z_1$ after multiple iterations approaches infinity.

$Z_2$ after multiple iterations is still on the unit circle

Figure 3.1 Basins of Attraction in the Complex Plane

is in the basin of attraction of $\infty$. The boundary between the two basins of attractions, for our example the unit circle, is a Julia set.

If we pick a C other than 0. the boundary is not a circle, and there can be attractors other than 0 and $\infty$. In such cases, there is competition among several attractors for domination of the plane. The boundaries are seldom simple, instead they are an "unending filigreed entanglement and unceasing bargaining" for control of the plane [Ref. 5: p. 4]. The border regions are an area of transition, from one domain of attraction to another. It is here in the border regions that the interesting pictures lay waiting.

There are two major classes of Julia sets. Some are in one piece and we call them connected. The others are not in one piece, they are a cloud of points called Cantor sets. Adrien Douady has given the name *Mandelbrot Set* to the set of all values of C (in the function $Z_{n+1} = Z_n^2 + C$ ) for which the Julia sets are connected. [Ref. 5: p. 161]

## D. THE MANDELBROT SET

The Mandelbrot set is a graphical representation of the values of C for which connected Julia sets result from the iteration of $Z_{n+1} = Z_n^2 + C$ [Ref. 6: p. 16]. The Mandelbrot set has a cardoid shaped main body with multiple smaller buds growing from the body. The Mandelbrot set is the black portion in the center of the plane (Figure 3.2). The rings around the Mandelbrot set are not part of the set, though they are usually the prettiest part of the pictures (Figure 3.3).

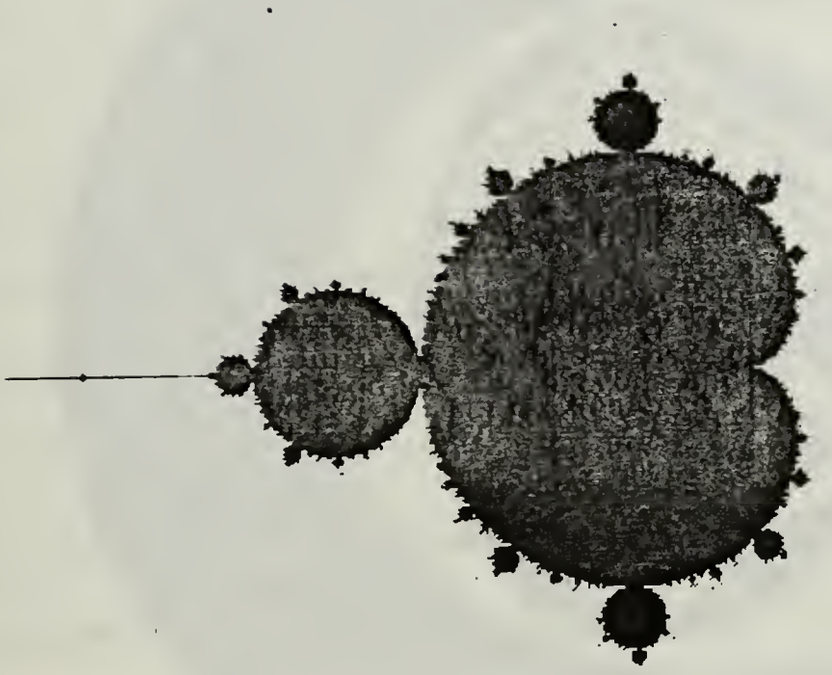


Figure 3.2 The Mandelbrot Set

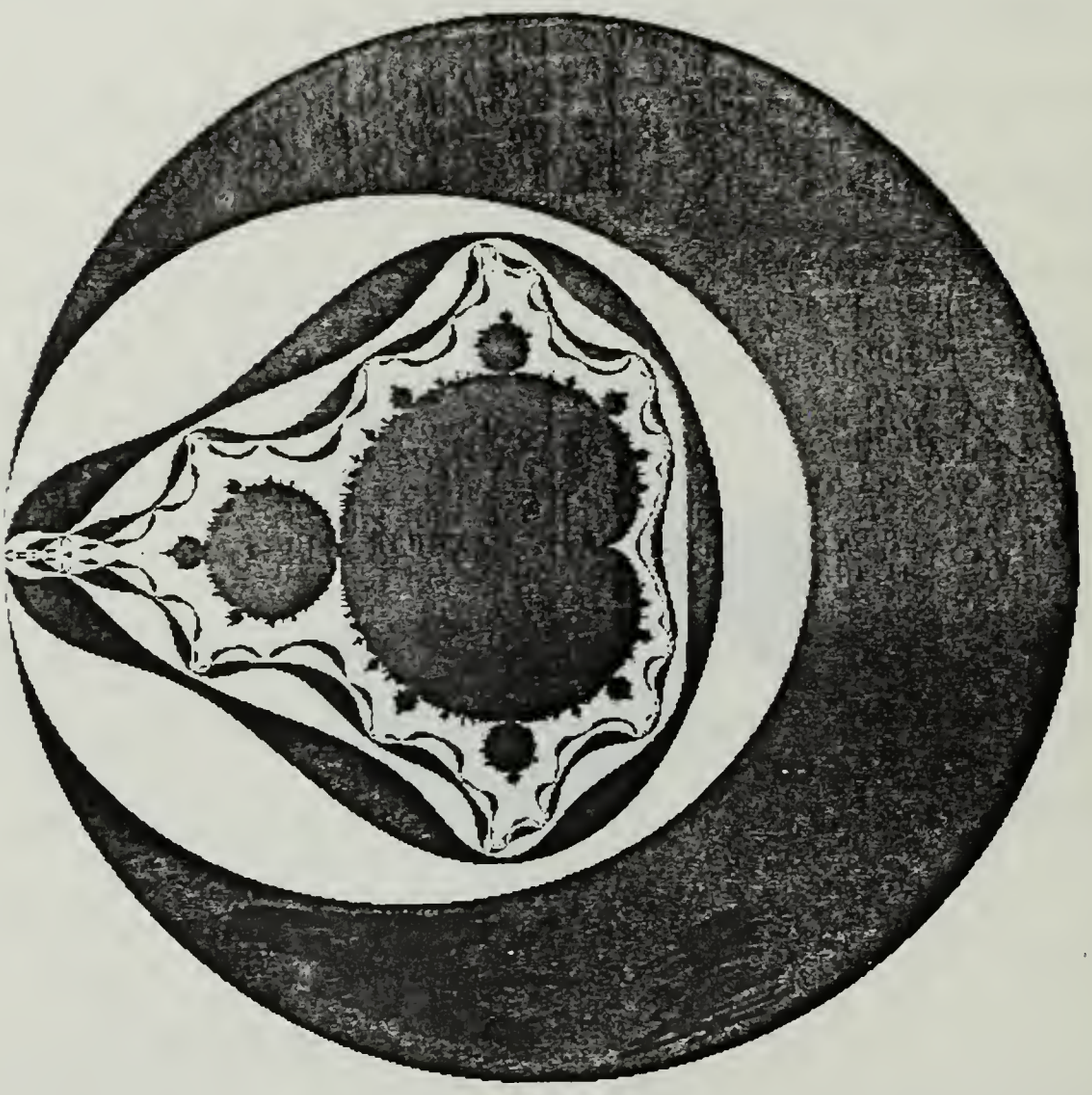


Figure 3.3 The Mandelbrot Set with Rings

The rings represent an escape time towards $\infty$. Mathematicians have shown that if the modulus of **Z** ever exceeds two then **Z** is in the basin of attraction of $\infty$ [Ref. 7: p. 77]. Escape time is a measure of how fast **Z** is moving toward $\infty$, and is calculated by counting the number of iterations required to reach a modulus of two. If a modulus of two is not reached within a set maximum number of iterations, the point is considered to be in the Mandelbrot set. Most of the pictures of Julia sets and the Mandelbrot set are obtained by computing the escape time for each point of the screen, and coloring the point accordingly using a color map calculated for maximum aesthetic effect !

If a value of C is picked outside the Mandelbrot set, a disconnected Julia set results. If a value of C is picked in the Mandelbrot set, a connected Julia set results. There are four main types of Julia sets. C's position in the Mandelbrot set determines what type of Julia set is computed.

- "If C is in the interior of the main body of the Mandelbrot set, a fractally deformed circle surrounds one attractive fixed point.
- If C is in the interior of one of the buds, then the Julia set consists of infinitely many fractally deformed circles which surround the points of a periodic attractor and their pre-images.
- If C is the germination point of a bud, we have a parabolic case: the boundary has tendrils that reach up to the marginally stable attractor.
- If C is any other boundary point of the cardoid or a bud . . . we have a seigel disk." [Ref. 5: pp. 12-14]

A seigel disk is an invariant circle inside a Julia set. That is, if we take a point on the circle, all iterated points derived from that point will also be on

the circle. The type of Julia set determines the characteristics of the picture that is produced.

## E. MAKING PICTURES OF JULIA SETS AND THE MANDELBROT SET

To create the Mandelbrot set pictures we use the function $\mathbf{Z_{n+1} = Z_n^2 + C}$ in an algorithm summarized as follows:

**Loop1** until all pixels are calculated
  Set $Z_0 = 0 + 0i$.
  Set $C$ to the value of the complex plane for this pixel.
  Initialize a counter to 1.
  Set a maximum number of iterations ( 100 is a good place to start).
  Calculate $Z_{n+1} = Z_0^2 + C$.
  Calculate the modulus of $Z_{n+1}$.
  **Loop2** until the modulus of $Z_{n+1} \geqslant 2$ or the counter is $>$ your maximum number of iterations then
    Set $Z_n = Z_{n+1}$.
    Calculate $Z_{n+1} = Z_0^2 + C$.
    Calculate the modulus of $Z_{n+1}$.
    Add one to your counter.
  **End Loop2**
  Establish the color of the pixel based on the counter.
**End Loop1**

We establish a correspondence mapping the region of the complex plane for which we are calculating values to the pixels of the screen. Also, an escape radius is needed. That is, we need a modulus value. This value is used as a terminating condition for the recursive iteration, $\mathbf{Z_{n+1} = Z_n^2 + C}$ . If the modulus of $\mathbf{Z_{n+1}}$ exceeds this maximum modulus (escape radius) the current iteration is halted and a value for the color is assigned. If after multiple iterations the modulus ever

reaches the escape radius, the point $Z_0$ is considered to escape from the Mandelbrot set to the attractor at $\infty$. We use the value 2 for our escape radius. Chapter six includes some figures produced using the summary algorithm listed above.

To create Julia set pictures, the same procedure is followed except values of $C$ are picked from the Mandelbrot set and fixed at that value. $Z$ is then allowed to vary over the range of the complex plane corresponding to the pixel space. (Appendix A contains a program written in the C language that produces the Mandelbrot set.)

Most of the monster-like fractal pictures are produced creating Julia sets. They create pretty pictures, and represent one of the most complicated ideas in mathematics today. Julia sets are being studied in such diverse areas as the dynamics of chaos, and for calculating the roots of equations using Newton's method in the complex plane. Readers desiring more information on the uses of Julia sets and the Mandelbrot set are directed to [Ref. 5].

# IV. IMPLEMENTATION DETAILS: 2D INTERACTIVE FRACTAL WORKSHOP

## A. BACKGROUND

The program we have implemented is an interactive computer graphics workshop for two-dimensional fractals. More specifically, it allows experimentation with two-dimensional Koch-like curves. The program was started by Mike Gaddis in 1985, and our goal was to take the partial program and make it a functional program. In addition to making the program functional, we added some capabilities and made the program more robust. The normal steps involved in executing the program consist of building a generator, or selecting a previously constructed generator from a file. Then an initial object, composed of one or more initiators, is selected and the fractal curve is constructed by recursively replacing all initiators with generators. The results can be saved for a laser printer and the process can be started again.

## B. THE MENU ORGANIZATION

The program is a menu driven system with selections made through the use of a mouse. Our mouse is mobile and has three selection buttons on the top. Menu selections are made by holding down the middle mouse button and scrolling through the selections. When the desired selection is highlighted, the middle mouse button is released and that selection is executed. Throughout the program

there are on screen displays of the function of each mouse button. The following is a menu by menu description of our program's organization.

1. Main Menu

The first menu displayed on startup is titled MAIN MENU and is shown in Figure 4.1. There are six active options in the Main Menu.

The first option in the menu leads to the construction of a new generator. It is the main feature allowing the rapid experimentation with Koch-like curves. This mode enables the use of the mouse for the creation of a generator for use in the recursive substitution of the Koch-like fractal curves. This selection displays a menu ( Interactive Generator Build, Figure 4.2).

The second option of the Main Menu retrieves a previously created generator from a file. This mode is an alternative to redrawing the generator every time the program is run.

The third option in the Main Menu retrieves an initial object, consisting of one or more initiators, from a file. For the current system, there are 5 initial objects: a line, a square, a triangle, an "inverse square", and an "inverse triangle". The "inverse square" is defined by the same points as the square but the points are taken in reverse order. This causes the recursive replacement to be made on the opposite side of the line. (Figures 4.3 and 4.4) Similarly, the "inverse triangle" is defined by the same points as the triangle but the points are used in reverse order. The files for the initial objects are ASCII character files to allow for ease in changing the objects with a text editor.

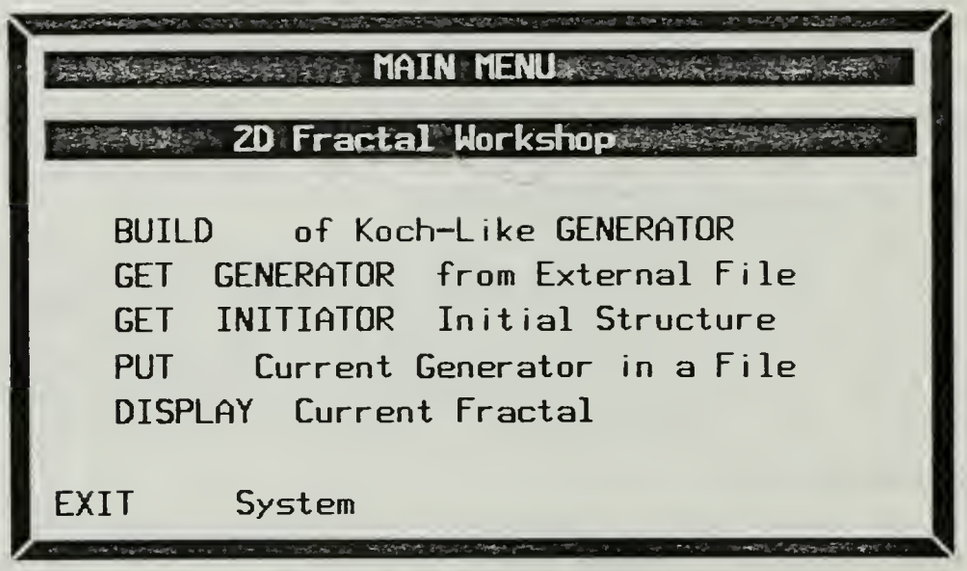



Figure 4.1 Main Menu

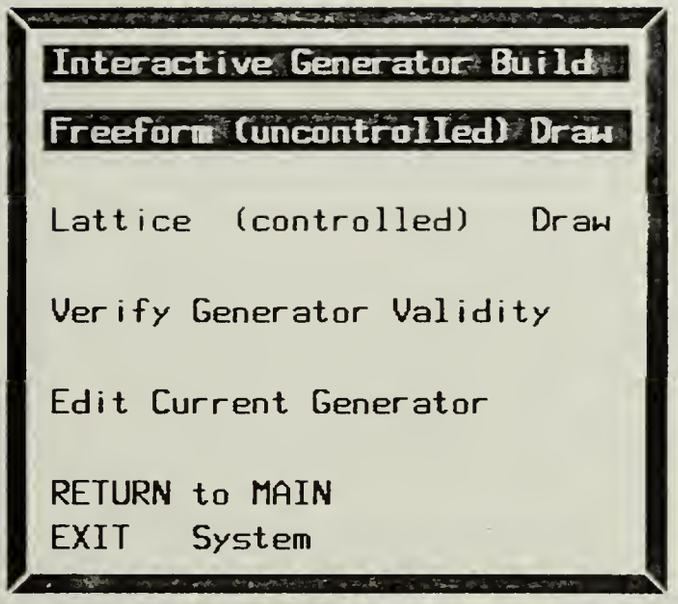
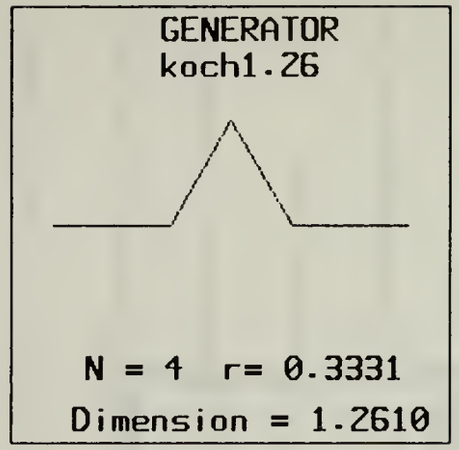
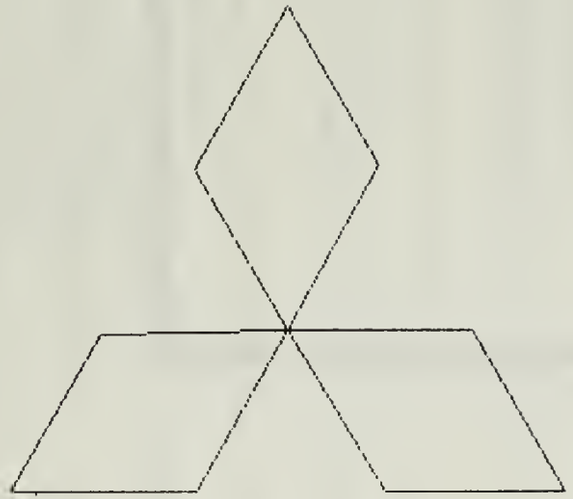



Figure 4.2 Interactive Generator Build Menu

Figure 4.3 Triangle







Max level of Recursion = 1

Figure 4.4 Inverted Traingle

The fourth option of the Main Menu saves a generator to a file for later use. The name of a file is requested, and the file is saved in the directory GENS, a subdirectory under the working directory.

The fifth option of the Main Menu is the first step in generating the fractal curve. Selecting this option causes the Fractal Curve Generation Menu (Figure 4.5) to be displayed.

Accessing the files for reading both the generators and initial objects is through a scroll and select process. The files for the generators are stored in the directory "GENS". The files for the initial objects are stored in the directory "OBJS". These directories must be direct subdirectories of the same directory containing the fractal workshop program. The directory containing the fractals workshop must be the current directory. The directory contents are displayed in a scroll and select menu.

2. Interactive Generator Build Menu

The Interactive Generator Build menu is retained primarily for later expansion of the program (Figure 4.2). It is displayed as an intermediate step on the way to building a generator. Currently there is only one active choice.

Selection 1 is the option for the interactive construction of a generator. This selection displays another menu, the Free Form Options menu (Figure 4.6). The *Return to Main* selection returns control to the Main Menu. The *Exit system* selection terminates the program and clears the graphics screen.

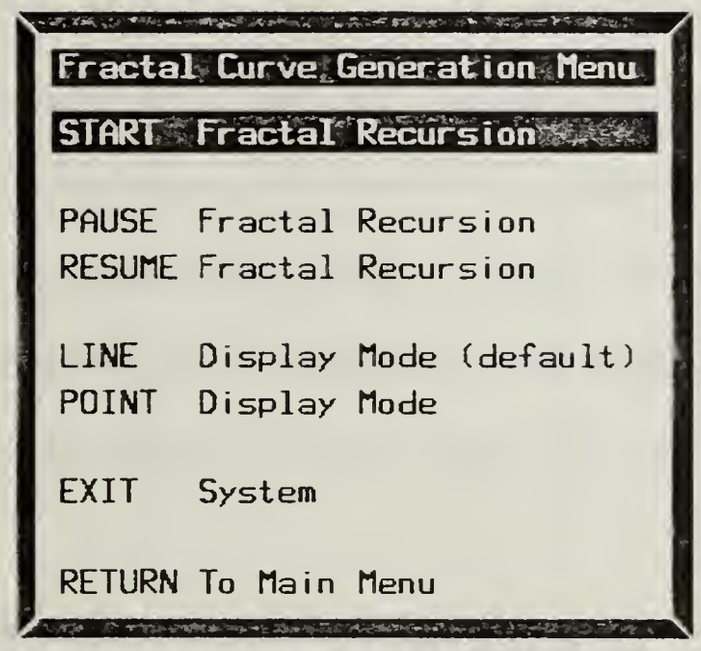



Figure 4.5 Fractal Curve Generation Menu

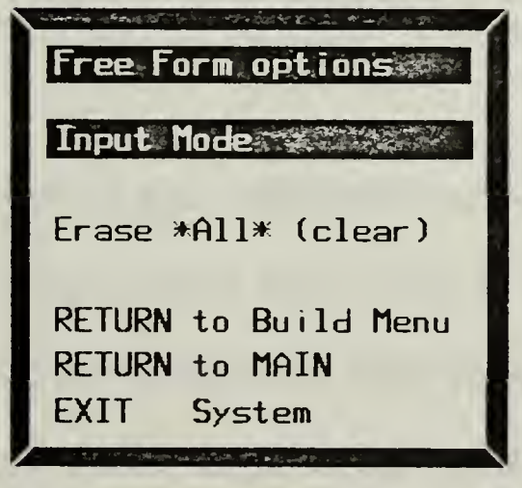



Figure 4.6 Free Form Options Menu

3. Free Form Options Menu

The Free Form Options menu is the last step on the way to building a generator. This menu has 5 options. The *Input Mode* option displays the screen for constructing the generator. The *Erase All* option erases the current generator from active memory. The *Return to Build Menu* option returns control to the Interactive Generator Build menu. The *Return to MAIN* selection returns control to the Main Menu. The *Exit System* selection terminates the program and clears the graphics screen.

4. Fractal Curve Generation Menu

The "Fractal Curve Generation" menu controls the creation and display of the fractal curve. There are seven active selections in this menu.

*Start Fractal Recursion* recursively replaces each initiator of the selected initial object with the selected generator, creating the fractal curve. When the fractal curve is completed, the Re-Run or Dump Bitmap menu is displayed (Figure 4.7).

*Pause Fractal Recursion* temporarily stops the execution of the program. The display on the screen is frozen until another option is selected. *Resume Fractal Recursion* restarts the program at the point where execution was halted.

*Line Display Mode* is the default display mode. This causes each initiator line to be erased and the new generator to be drawn in fully as line segments during the fractal curve generation.

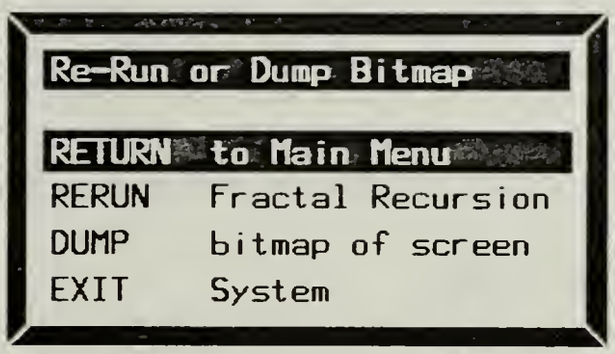



Figure 4.7 Re-Run or Dump Bitmap Menu

*Point Display Mode* displays only the end points of the generator's line segments as they are calculated. The generator line segment end points become the end points for new initiators. As long as the new end points are farther apart than the minimum recursion termination distance, replacement generator points are calculated and displayed between these new end points. Since the initiator line segments are not drawn, there is no need to erase them prior to replacing the initiators with generators. The final picture is similar to the line display mode, but the display during generation of the curve is different.

*Return to Main Menu*, causes control of the program to return to the Main Menu at the completion of the fractal curves generation. The fractal curve drawn is erased, but the initial object and generator are retained in memory.

5. Re-Run or Dump Bitmap Menu

The Re-Run or Dump Bitmap menu is displayed at the completion of the generation of the fractal curve. (Figure 4.7). There are four options in this menu. *Return to Main Menu* returns control of the program to the top level of the program. The fractal curve drawn is erased, but the initial object and the generator are retained in memory. *Rerun Fractal Recursion* returns control to the Fractal Curve Generation menu (Figure 4.5), and the curve created is erased from the screen. Both the initial object and the generator are retained in memory. *Dump bitmap of screen* asks for a file name in which to save the bitmap. This selection reads the pixels of the screen and saves the information necessary such that the file can be printed on a laser printer.

# V. BUILDING A GENERATOR

## A. HOW TO BUILD A GENERATOR

A generator is the set of line segments that is used to replace an initiator. To build a generator. a rectangular part of the screen is set up as a drawing area. The initiator end points are displayed at the left and right edges of the drawing area. along with a pencil shaped cursor that moves within the drawing area. Generator size is limited by the size of this rectangle. The end points of the generator are fixed coincident with the end points of the initiator. Having the end points fixed makes it easier to calculate the other points that define the generator. Since the end points of the initiator and the generator are coincident, and the end points for the initiator are always known, data is not saved for the end points of the generator being constructed. The data structure that saves the information defining the generator has room for 20 points. Therefore, the maximum number of line segments in a generator is 21.

When the mouse is moved, in addition to the pencil cursor moving, a new line is drawn from the last saved point (at the start it is the generator left end point) to the current cursor position. Another line is drawn from the current cursor position to the generator right end point. These two lines act as a "rubber band"

line that follows the cursor around the drawing rectangle. This allows the viewing of potential generator segments without actually designating points.

The three buttons on the mouse are used to control options for building the generator. When the middle mouse button is pressed, the current cursor position is saved and becomes a point in the generator. The saved point becomes the new start of the "rubber band" line. The left mouse button erases the last point added to the generator. The right mouse button says that the generator is complete and control is returned to the Free Form Options menu. The final generator is composed of a set of line segments. The line segments are drawn consecutively starting at the left end point. to the first saved point. then the second saved point, etc. until the last saved point is reached. The final line segment is drawn from the last saved point to the right end point. The line segments from the last saved point to the cursor position and then to the right end point are not included in the generator.

Figure 5.1 is a partial screen bitmap, and indicates that information is being saved for three generator points. The small picture of the generator indicates that the two line segments from the last saved point to the current cursor position, and then to the right end point are not included in the generator.

1. What information is saved for a Generator?

In screen coordinates, the drawing rectangle is 200 units in the X direction, and 250 units in the Y direction. The size of a generator is restricted by

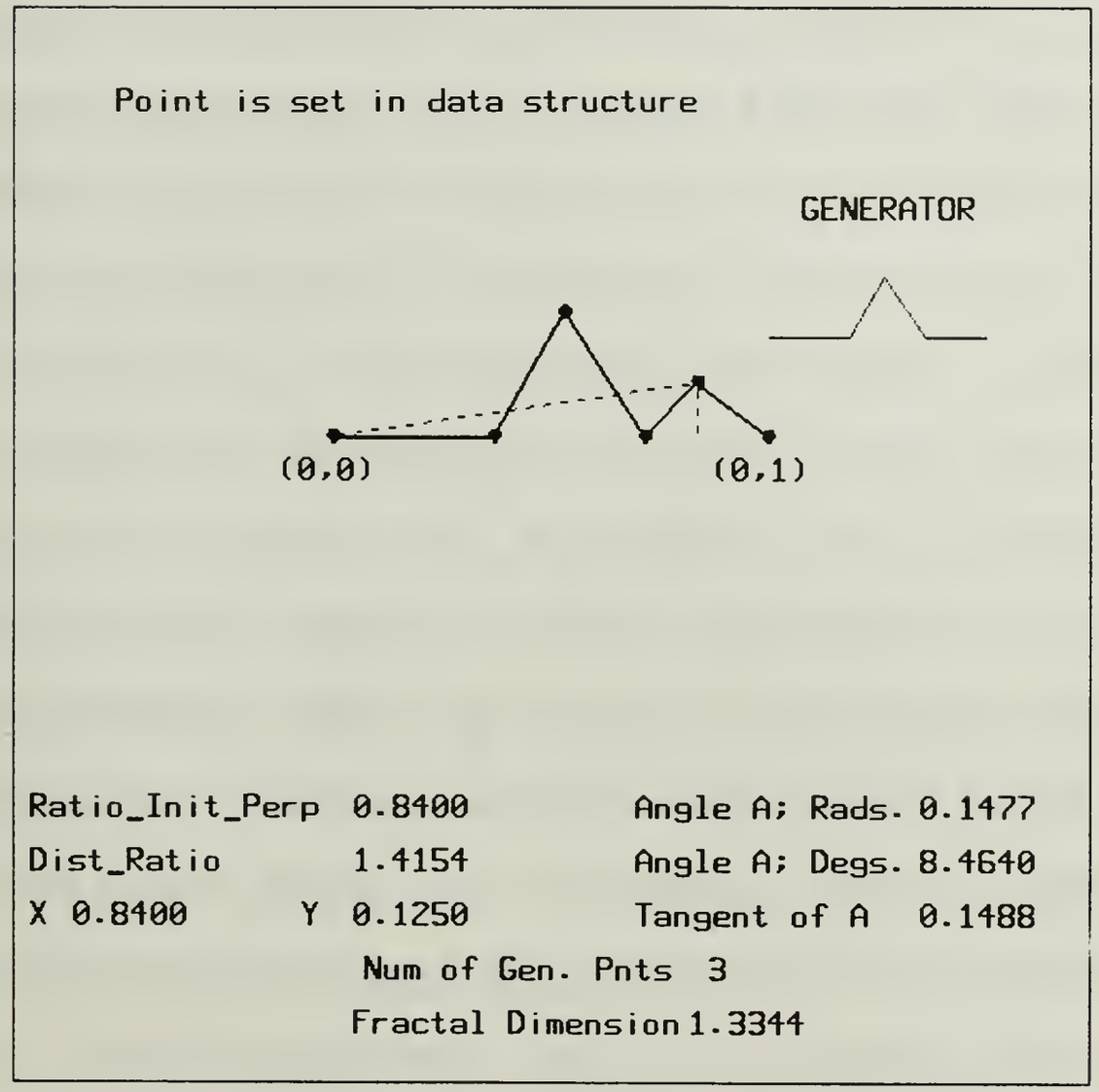



Figure 5.1 Building a Generator

this drawing rectangle. The distance between the generator end points is 200 units. This information is used in determining information to define new generator points.

It is easy to determine the X and Y coordinates for the saved points from cursor position. The X and Y coordinates are then used to calculate information necessary to construct a generator from any two arbitrary end points defining an initiator. Having the fixed X, Y coordinates for a generator point is not enough information to determine where that generator point is to be placed for an initiator that is different in size and orientation from the fixed situation of the generator build. What is needed is the ability to determine the relative positioning of the generator point, given only two arbitrary initiator end points.

The relative measures we use are a ratio of where the generator point is located between the two initiator end points, and angles to determine how far away from the initiator the generator point is to be placed. We move a line that is perpendicular to the initiator from the first end point towards the second end point until the generator point lies on the perpendicular line. We save a ratio of the distance the perpendicular line has moved when the generator point is on the perpendicular line, to the distance between the initiator end points. Knowing this ratio is still not sufficient information to determine where the generator point is to be placed. The generator point can lie anywhere along the perpendicular line. If we use only X and Y coordinates and the initiator line is moved to another position in the plane, the X and Y coordinates are not valid for the new generator

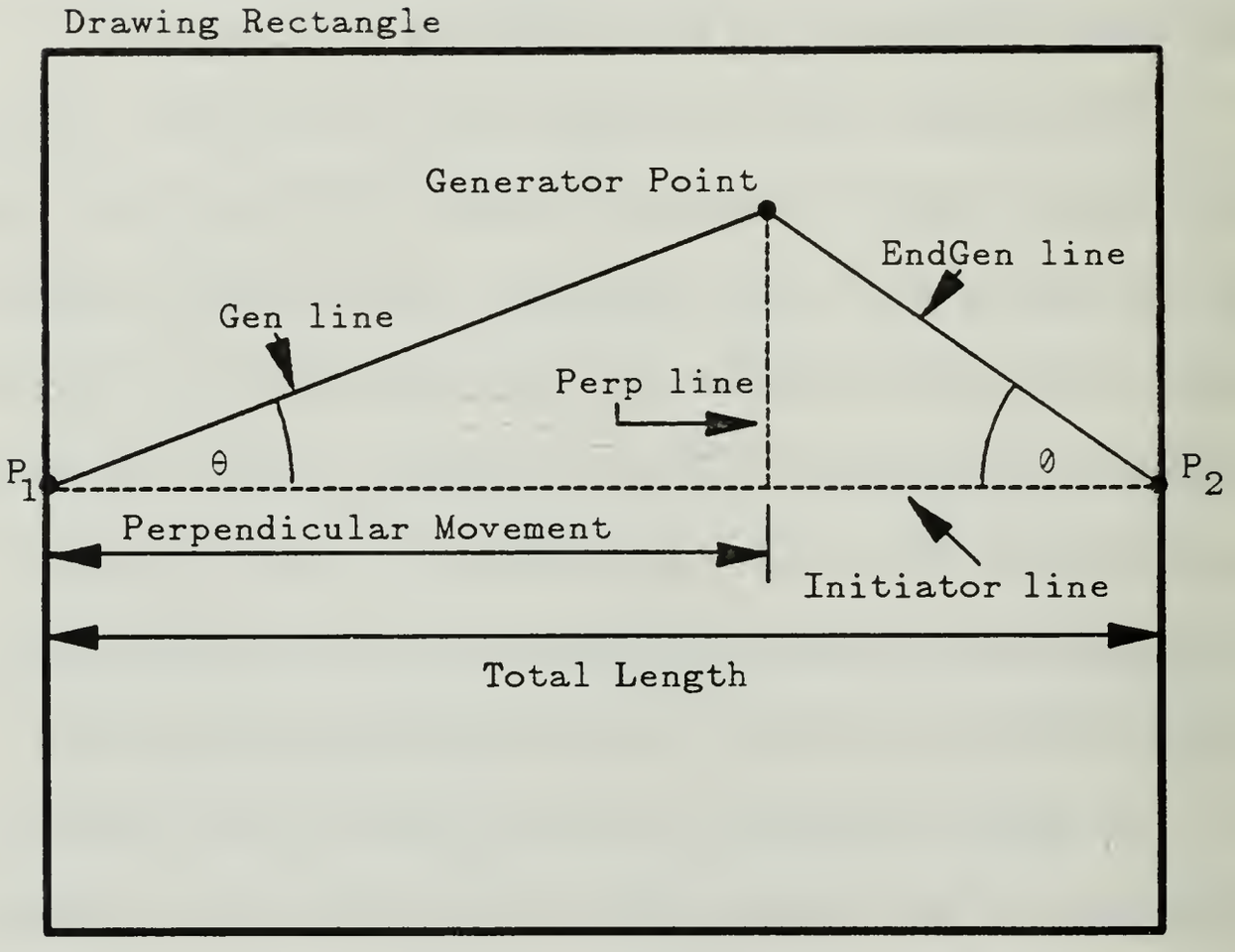
point. If we use the distance from the generator point to the initiator along the perpendicular line, that distance is not valid for an initiator that is longer or shorter than the fixed size generator build initiator. We need to know a relative positioning to determine where to place the generator point. To discriminate where along the perpendicular line the generator point lies we use angles.

For calculating where the generator point is to be placed. we use a relative distance along the perpendicular line, given that we know the distance from the first end point to the perpendicular line. This relative measure. a distance along the perpendicular over the distance from the first end point to the perpendicular is the tangent of the angle formed by the initiator and the line connecting the first end point and the generator point. We call this angle $\theta$. We also calculate the angle formed by the initiator and the line connecting the second end point and the generator point. We call the second angle $\phi$ (Figure 5.2).

To define a generator point, we save a ratio of movement between the end points and we save information about the two angles $\theta$ and $\phi$. One angle is sufficient in most situations to determine where the generator point is to be placed. We save two angles to handle special cases. The data that we store in memory to represent an angle is the tangent of that angle.

2. Data Structure for the Initial Object

The initial objects are represented by a set of points (X, Y coordinates), and a number representing the number of points in the object. An object can be represented by up to 25 points. The object is created by starting at point one and




θ = Angle between Gen line and the initiator

∅ = Angle between EndGen line and the initiator

$$\text{ratio} = \frac{\text{Perpendicular Movement}}{\text{Total Length}}$$

Figure 5.2 Defining A Generator Point

drawing a line segment to point two, then drawing a line segment to point three, etc. This procedure is carried out until the last point in the object has been reached. A closed object requires a repeated point. For example, a triangle requires four points, with the first and the fourth point the same.

## B. CREATING THE FRACTAL CURVE

### 1. Fractal Curve Generation Overview

The program for generating the fractal curve is both iterative and recursive. We assume that an object is composed of one or more initiators (line segments). Each of the original initiators is dealt with in the order they are defined in memory. The length of the initiator is calculated. If the initiator length is greater than a specified minimum distance, a replacement generator is calculated for that initiator. This replacement generator is composed of new initiators, and they are handled consecutively. If the length of the new initiator is greater than the minimum distance, the process is repeated recursively. The terminating condition for the recursion is the length of an initiator being less than or equal to the minimum distance. We use one pixels width for a minimum distance. When the recursion terminates, the next initiator at the current level is examined. This continues until all the initiators of the original object have been considered.

The heart of the program is an algorithm that calculates the points for a replacement generator. Using this algorithm, given any two arbitrary points

representing the end points of an initiator. a set of generator points is determined. These generator points are scaled and oriented such that a generator replaces the initiator. The calculations are not complex. They are basic trigonometry and geometry calculations. However. they are carried out many times.

2. The Algorithm for Calculating the Generator Points

This algorithm assumes that the X, Y coordinates of the initiator end points are known. Since the end points of the initiator and the replacement generator are coincident, they are referred to only as end points. The general method for determining the new generator uses the X and Y coordinates of the two end points and the ratios and angles information that defines the generator. Using this information, we find the equations, in terms of the X and Y values of the generator point, of two different lines passing through the generator point. We solve these two equations to find the values of the two unknowns, the X and Y of the generator point. Once the X and Y coordinates have been determined, the initiator is erased and the lines for the generator are drawn. The algorithm being described here does not do all of the checking for special cases that a functional program requires. An example of a special case that needs to be handled is the prevention of division by zero in an equation. The function that the workshop uses is included in Appendix B, and does the necessary special case checking. Figure 5.3 summarizes the steps of the algorithm.

**Repeat** for all generator points.
Calculate the distance between end points.
Calculate the slope of the initiator.
If saved ratio for this generator point does not equal 0 then
**begin** if
Calculate the point of intersection, $P_{perp}$, of the line, Perp, perpendicular to the initiator and through the generator point.

If $\theta = 0$
then the generator point is $P_{perp}$.
else
**begin** else
Calculate $Slope_{gen}$ of the line, Gen, through the first end point and the generator point.
Calculate the Y intercept of the line Gen.
$Y_{unknown} = Slope_{gen} \times X_{unknown} + b_{gen}$.
Calculate **Slopeperp**.
Calculate the Y intercept of the line Perp.
$Y_{unknown} = Slope_{perp} \times X_{unknown} + b_{gen}$.
**end** else
**end** if ratio not = 0
else ratio = 0
**begin** else
Calculate $Slope_{gen}$ for the line, Gen, through the first end point and the generator point.
Calculate the Y intercept for the line Gen.
$Y_{unknown} = Slope_{gen} \times X_{unknown} + b_{gen}$.
Calculate $Slope_{endgen}$ for the line, endgen, through the second end point and the generator point.
Calculate the Y intercept for the line endgen.
$Y_{unknown} = Slope_{endgen} \times X_{unknown} + b_{endgen}$.
**end** else ratio = 0
Solve the two equations and determine a value for $Y_{unknown}$.
Use the value for $Y_{unknown}$ in one of the two equations and solve for $X_{unknown}$.
**End** Repeat.

All the X and Y values for the generator point are now known.

Figure 5.3 Algorithm for Calculating Generator Points

Unlike the von Koch snowflake that can theoretically continue it's replacements an infinite number of times, we use a finite machine and need a termination condition for the replacements. We stop the recursion when the distance between two end points (any arbitrary initiator) is less than the distance that can be displayed in one pixel on our viewing screen. The first step is to determine the distance. **DIST**, between the two points. For simplicity we label the first end point $P_1$, and label the second end point $P_2$. If **DIST** is less than a specified minimum distance. the recursion stops. However, if DIST is greater than the terminating distance, the points for a generator are calculated to replace the initiator between the points $P_1$ and $P_2$.

The steps to calculate the X and Y values for each of generator points are similar. We determine the slope of the line connecting points $P_1$ and $P_2$,and call it the slope of the initiator, $\mathbf{Slope_{init}}$.

$$\mathbf{Slope_{init}} = \frac{Y_2 - Y_1}{X_2 - X_1}$$

If the ratio of movement between points $P_1$ and $P_2$ is not **0** then we calculate the point of intersection on the initiator of a line through the generator point that is perpendicular to the initiator. This point is called the perpendicular point and has values $X_{perp}$ and $Y_{perp}$.

$$X_{perp} = X_1 + savedRatio \times (X_2 - X_1)$$

$$Y_{perp} = Y_1 + savedRatio \times (Y_2 - Y_1)$$

Then the angle $\theta$ is used to determine where along the perpendicular line the generator point lies. If $\theta = 0$ the point is on the initiator and the X and Y values are $X_{perp}$ and $Y_{perp}$. If $\theta$ is not 0, the X and Y values must be calculated. To do this, we must determine the slope of the line connecting the point $P_1$ and the generator point, call it $Slope_{gen}$. We now use the knowledge that the slope of a line is also the tangent of the angle formed by that line and the X axis. We know $Slope_{init}$ and we know $\tan\theta$. We use $Slope_{init}$ as the tangent of the angle $\gamma$, the angle formed by the initiator and the X axis. We determine the tangent of the sum of the two angles, $\gamma$ and $\theta$. $Tan(\gamma + \theta)$ is the slope of the line connecting point $P_1$ and the generator point, $Slope_{gen}$. Figure 5.4 indicates how $\gamma$ and $\theta$ are positioned to allow us to use the tangent of the sum of two angles to determine $Slope_{gen}$.

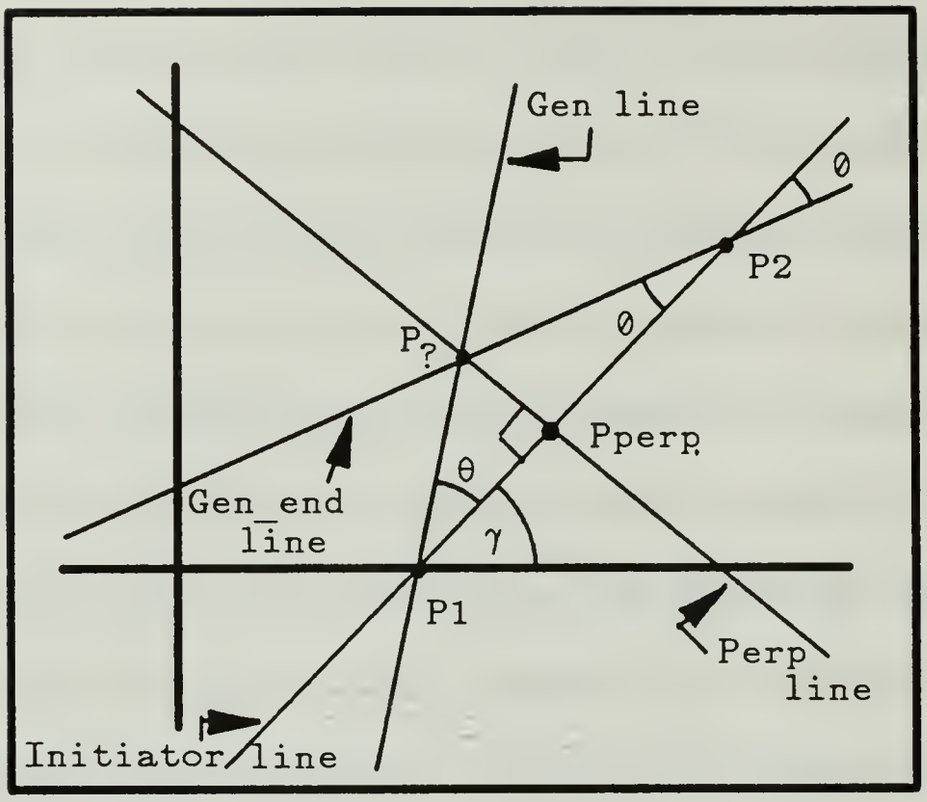
$$Slope_{gen} = \frac{\tan\theta + \tan\gamma}{1.0 - \tan\theta \times \tan\gamma}$$

We now find the Y intercept, $b_{gen}$, for the line through $P_1$ and the generator point.

$$b_{gen} = Y_1 - Slope_{gen} \times X_1$$

This gives us an equation for the line through point $P_1$ and the unknown generator point.

$$Y_{unknown} = Slope_{gen} \times X_{unknown} + b_{gen} .$$




$$\text{Slope}_{\text{initiator}} = \tan \gamma$$

$$\text{Slope}_{\text{Gen}} = \tan (\theta + \gamma)$$

$$\text{Slope}_{\text{Gen-end}} = \tan (\gamma - \emptyset)$$

Figure 5.4 The Angles and Lines defining the Generator

We also know that the slope of the line perpendicular to the initiator is

$$\mathbf{Slope_{perp}} = \frac{-1.0}{\mathbf{Slope_{init}}}.$$

Using $\mathbf{Slope_{perp}}$ and the point $\mathbf{X_{perp}}$, $\mathbf{Y_{perp}}$ we can determine the Y intercept point for the perpendicular line.

$$\mathbf{b_{perp}} = \mathbf{Y_{perp}} - \mathbf{Slope_{perp}} \times \mathbf{X_{perp}}.$$

This gives us the equation for the perpendicular line through the unknown generator point.

$$\mathbf{Y_{unknown}} = \mathbf{Slope_{perp}} \times \mathbf{X_{unknown}} + \mathbf{b_{gen}}.$$

Thus we have the two equations with two unknown values, $\mathbf{X_{unknown}}$ and $\mathbf{Y_{unknown}}$. Solving these equations for $\mathbf{X_{unknown}}$ we get

$$\mathbf{X_{unknown}} = \frac{\mathbf{b_{perp}} - \mathbf{b_{gen}}}{\mathbf{Slope_{gen}} - \mathbf{Slope_{perp}}}.$$

Using this value for $\mathbf{X_{unknown}}$ in either of the two equations, we determine $\mathbf{Y_{unknown}}$.

$$\mathbf{Y_{unknown}} = \mathbf{Slope_{gen}} \times \mathbf{X_{unknown}} + \mathbf{b_{gen}}.$$

The X and Y coordinates for the generator point are now known.

If the ratio is 0, a different method is needed. If the ratio is 0, we know the generator point lies on a line perpendicular to the initiator and passing through the point $\mathbf{P_1}$. $\boldsymbol{\theta}$ is $\pm 90^{\circ}$ and will not define a unique generator point.

The slope of the line connecting the point $P_1$ and the unknown generator point is

$$\mathbf{Slope_{gen}} = \frac{-1.0}{\mathbf{Slope_{init}}}.$$

We now find the Y intercept, $b_{gen}$, for the line through $P_1$ and the unknown generator point.

$$b_{gen} = Y_1 - \mathbf{Slope_{gen}} \times X_1.$$

This gives us an equation for the line through point $P_1$ and the unknown generator point.

$$Y_{unknown} = \mathbf{Slope_{gen}} \times X_{unknown} + b_{gen}.$$

We now use the angle $\phi$. We determine the slope, $\mathbf{Slope_{endgen}}$, of the line connecting the point $P_2$ and the unknown generator point similarly to the calculation of $\mathbf{Slope_{gen}}$.

$$\mathbf{Slope_{endgen}} = \frac{\tan\theta - \tan\phi}{1 + \tan\theta \tan\phi}.$$

We now find the Y intercept, $b_{gen}$, for the line through $P_2$ and the unknown generator point.

$$b_{endgen} = Y_2 - \mathbf{Slope_{endgen}} \times X_2.$$

This gives us the equation for the line connecting the unknown generator point and point $P_2$.

$$Y_{unknown} = \mathbf{Slope_{endgen}} \times X_{unknown} + b_{endgen}.$$

Thus we have the two equations with two unknown values, $X_{unknown}$ and $Y_{unknown}$. Solving these equations for $X_{unknown}$ we get

$$X_{unknown} = \frac{b_{endgen} - b_{gen}}{Slope_{gen} - Slope_{endgen}}.$$

Using this value for $X_{unknown}$ in either of the two equations, we determine $Y_{unknown}$.

$$Y_{unknown} = Slope_{endgen} \times X_{unknown} + b_{endgen}.$$

The X and Y values for the generator point are now determined. The process is repeated to determine the X and Y values for the next generator point. This is continued for all the generator points.

When all of the points are known, we replace the initiator with a new generator. A line is drawn from point to point, in order, creating the generator with the proper orientation and scaling. The recursion is continued using the calculated points as the end points for the next level of initiators. When all of the initiators have been replaced to their lowest levels, the picture is done. Control is passed to the Re-Run or Dump Bitmap menu.

# VI. WORKSHOP EXPERIMENTS CONDUCTED

## A. INTRODUCTION

We have now introduced two programs that construct Koch-like fractal curves and the Mandelbrot set. It is time to show some of the results that have been obtained from these programs. Figures 1.1, 1.2, 1.3 and 2.5 discussed above are products of our experimentation. Some of our experiments were conducted to verify a property of the program. i.e., dimension calculations. Some were conducted to construct "pretty" pictures and some were conducted just to see what would happen. An interactive workshop allows this flexibility.

## B. DIMENSION VERIFICATION FOR THE FRACTAL CURVES

The equation we use for the fractal dimension in the generation of fractal curves is

$$\textbf{Fractal Dimension} = D = \frac{\log N}{\log \left(\frac{1}{r}\right)}.$$

N is the number of line segments in the generator and r is the scaling ratio for these line segments. This produces excellent results for the fractal dimension when all the line segments that compose the generator are equal in length. When their lengths are not equal, we calculate an average length for the line segments to

compute the scaling ratio. This causes our results to be an approximation for the fractal dimension when the line segment length varies.

There are several generators with equal length line segments that we use for dimension verification. The Koch generator is one of these. Since each line segment of the Koch generator is $\frac{1}{3}$ the original length and there are four line segments. the dimension should be

$$D = \frac{\log 4}{\log 3} \approx 1.26.$$

Figure 6.1 is the Koch generator used with a single initiator for constructing the fractal curve. It has a fractal dimension of 1.2610. Figure 6.2 is the same generator, with a triangle as the inititial object. This combination produces the Koch snowflake discussed earlier.

Figure 6.3 has a generator that is constructed of 8 line segments, each line segment has a scaling ratio of $\frac{1}{4}$. The dimension is

$$D = \frac{\log 8}{\log 4} = 1.5$$

When the generator with a fractal dimension of 1.5 is used with the line initial object, the fractal curve constructed is more "wiggly" than the fractal curve constructed with the Koch generator. We also create a variant of the generator with a fractal dimension of 1.5. The variant generator looks identical to the generator with a fracal dimension of 1.5, but it is constructed with seven line

Figure 6.1 Koch Line

Figure 6.2 Koch Triangle

Figure 6.3 1.5 Line

segments instead of eight line segments. Six of the line segments have a scaling ratio of $\frac{1}{4}$ and one line segment has a scaling ratio of $\frac{1}{2}$. The average scaling ratio is now .2857 and results in a higher fractal dimension, 1.5533. The resulting fractal curve (Figure 6.4) is even more "wiggly" than the fractal curve constructed with the generator that has a fractal dimension of 1.5 (Figure 6.3).

Another generator that is used to verify the fractal dimension is the generator constructed with 9 line segments, all with a scaling ratio of $\frac{1}{3}$. The fractal dimension should be

$$D = \frac{\log 9}{\log 3} = 2.0.$$

The generator in Figure 6.5 is constructed of nine line segments. Each line segment is about $\frac{1}{3}$ the length of the initiator. The minimum resolution for generator points in the workshop is .005 units, and $\frac{1}{3}$ is approximated with the value .335. We calculated that the generator in Figure 6.5 has a fractal dimension of 2.01, close to 2.0. The fractal curve almost fills an entire polygon (Figure 6.5). The inability of our program to construct the generator line segments exactly $\frac{1}{3}$ the original length is the reason for the gaps in the polygon, and the deviation from the dimension being exactly 2.0.

Figure 6.4 1.5 Variant Line

Figure 6.5 2.0 Line

## C. PRETTY KOCH-LIKE CURVES

We now turn to some of the prettier Koch-like curves we have constructed. We begin with a very simple generator and with only minor changes in the generators we get some drastic changes in the fractal curves.

We construct a simple generator of two equal length line segments forming an inverted 'v'. The resulting fractal curve constructed with a single line initial object is as expected, a simple almost cloud-like fractal curve (Figure 6.6). The result looks even more like a cloud when this generator is used with a square for an initial object (Figure 6.7).

The fractal curve in Figure 6.8 is another example of a fractal curve in which the result was as predicted. This fractal curve is included partially for sentimental reasons. This was the first fractal curve that we printed on the laser printer. With only a slight change in the generator, the results are a complete surprise (Figure 6.9).

Figures 6.10 and 6.11 are similar generators with only small perturbations in the line segments of the generator. The results are again surprisingly different, although quite pleasing to view. With the exception of Figures 6.2 and 6.7 all of the fractal curves within this chapter so far have been constructed from a single line segment initial object (one initiator). These curves constructed with the line as an initial object run faster, and are simpler than those curves constructed with an initial object composed of more than one initiator. Adding initiators adds complexity to the pictures (Figure 6.12).

Figure 6.6 A Line

Figure 6.7 A Square

Figure 6.8 First Line

Figure 6.9 Colleen Line

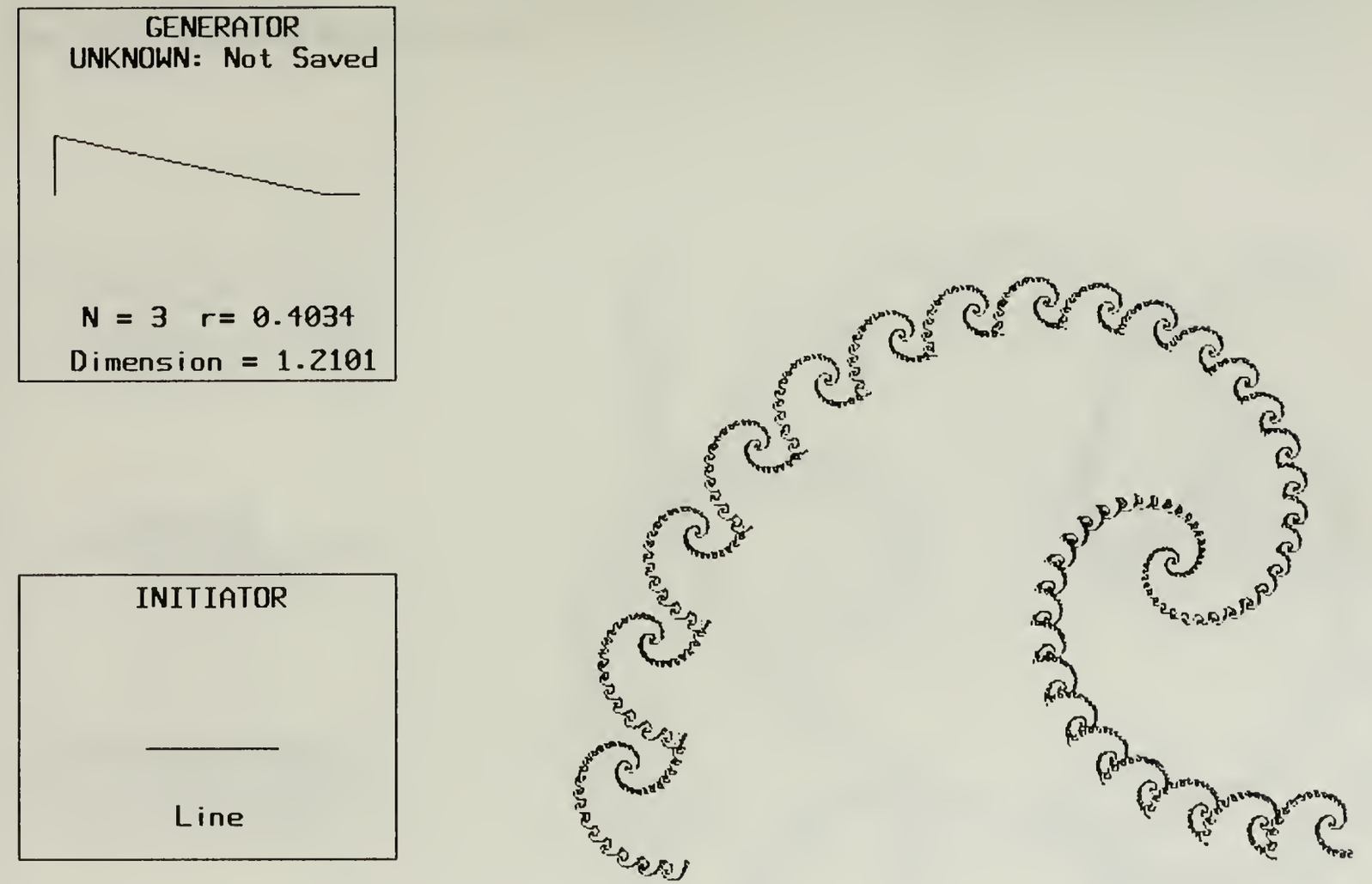



Figure 6.10 Wave Line

Figure 6.11 Spiral Line

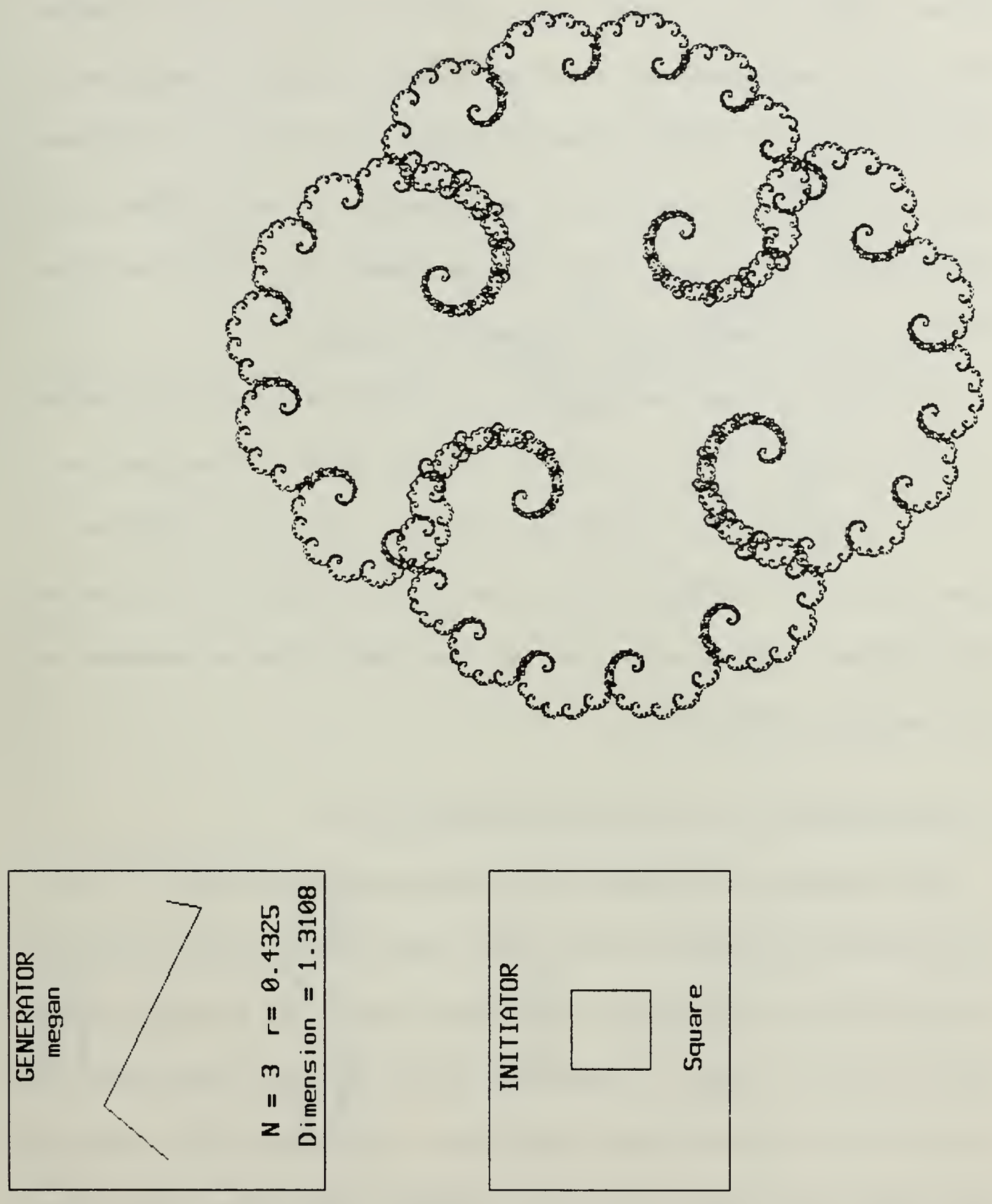



Figure 6.12 Meagan Square

Additional variations are achieved by using the "inverse triangle" and "inverse square" for the initial objects. Figure 6.13 is the Koch snowflake "inside-out". Sometimes though the difference between using the triangle and the "inverted triangle" is negligible. Figures 6.14 and 6.15 look the same. This shows that the symmetry in the generator sometimes negates the use of different initial objects. Figure 6.14 is constructed using the triangle initial object and Figure 6.15 is constructed using the "inverse triangle" initial object.

The ability to make small changes to a generator and watch the resulting fractal curve grow is both entertaining and informative. The resulting fractal curve is often not what is expected intuitively. Part of the enjoyment of experimenting with the fractal curves is the attempt to predict how the resulting curve will look. This is not an easy task, particularly when the generators are more complicated (Figure 6.16).

## D. EXPERIMENTS WITH THE MANDELBROT SET

The Mandelbrot set program we have implemented has an ability to "zoom" in and examine small sections of the complex plane in detail. With apologies, we present black and white replicas of the figures that on the computer graphics screen are alive in color. A large part of the time spent constructing the Mandelbrot set pictures is spent constructing a color scheme that presents the most aesthetically appealing picture. The black and white replicas do not do these pictures justice. To examine some outstanding color Mandelbrot and Julia sets

Figure 6.13 Koch Inverted Triangle

Figure 6.14 1.5 Triangle

Figure 6.15 1.5 Inverted Triangle

Figure 6.16 Anns Line

the interested reader is directed to [Ref. 5]. A list of regions in the complex plane that are used to make beautiful pictures can be found in [Ref. 5: pp. 193-196].

Figure 6.17 is the Mandelbrot set without any magnification, or any of the colored rings with which it is surrounded. Each colored ring represents a set of points that reach a modulus of 2 after the same number of iterations of the function $Z_{n+1} = Z_n + C$. Figure 6.18 includes some the colored rings. In Figure 6.18, we print every other colored ring for the first eight rings. That is, those rings where the points reaches a modulus of 2 after 2, 4, 6 or 8 iterations. The points that reach a modulus of 2 after 8 and before 100 iterations are left white. and are the portion adjacent to the Mandelbrot set. Figure 6.18 is a relatively high level picture and does not display much detail. The portion of the complex plane displayed is a rectangular region, from -2.0 to +2.0 along the real axis and from -2.0 to +2.0 along the imaginary axis. Figure 6.19 is the same portion of the complex plane, but with every other color up to 50 iterations displayed as black. Colors for 50 to 100 iterations are displayed as white.

Figure 6.20 is a close up of a small portion at the end of the "antenna" of the Mandelbrot set. The portion of the complex plane displayed is from -1.945 to -1.935 along the real axis, and from -0.005 to +0.005 along the imaginary axis. Every other color up to 50 iterations is displayed as black. Colors for iterations from 50 to 100 are left white. The black portion in the center of the figure is a part of the Mandelbrot set.

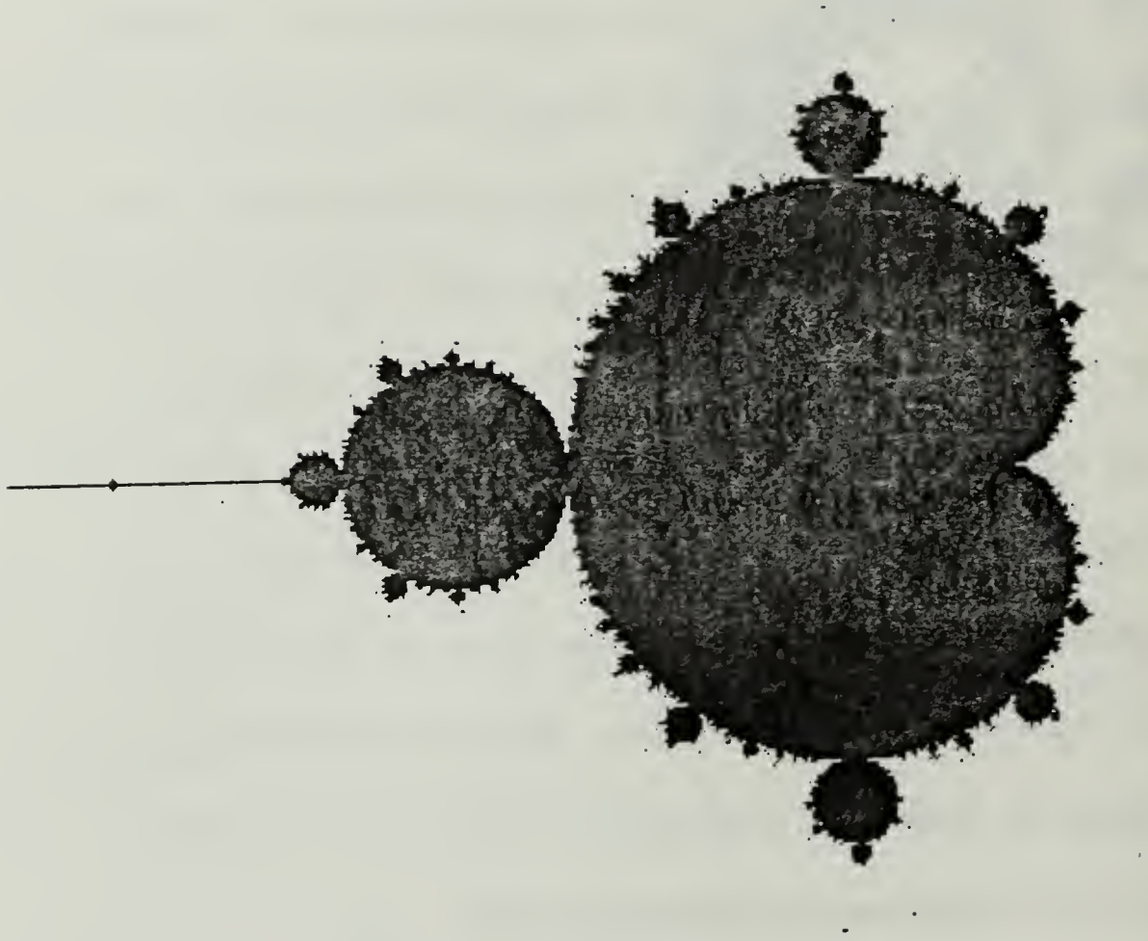


Figure 6.17 The Mandelbrot Set

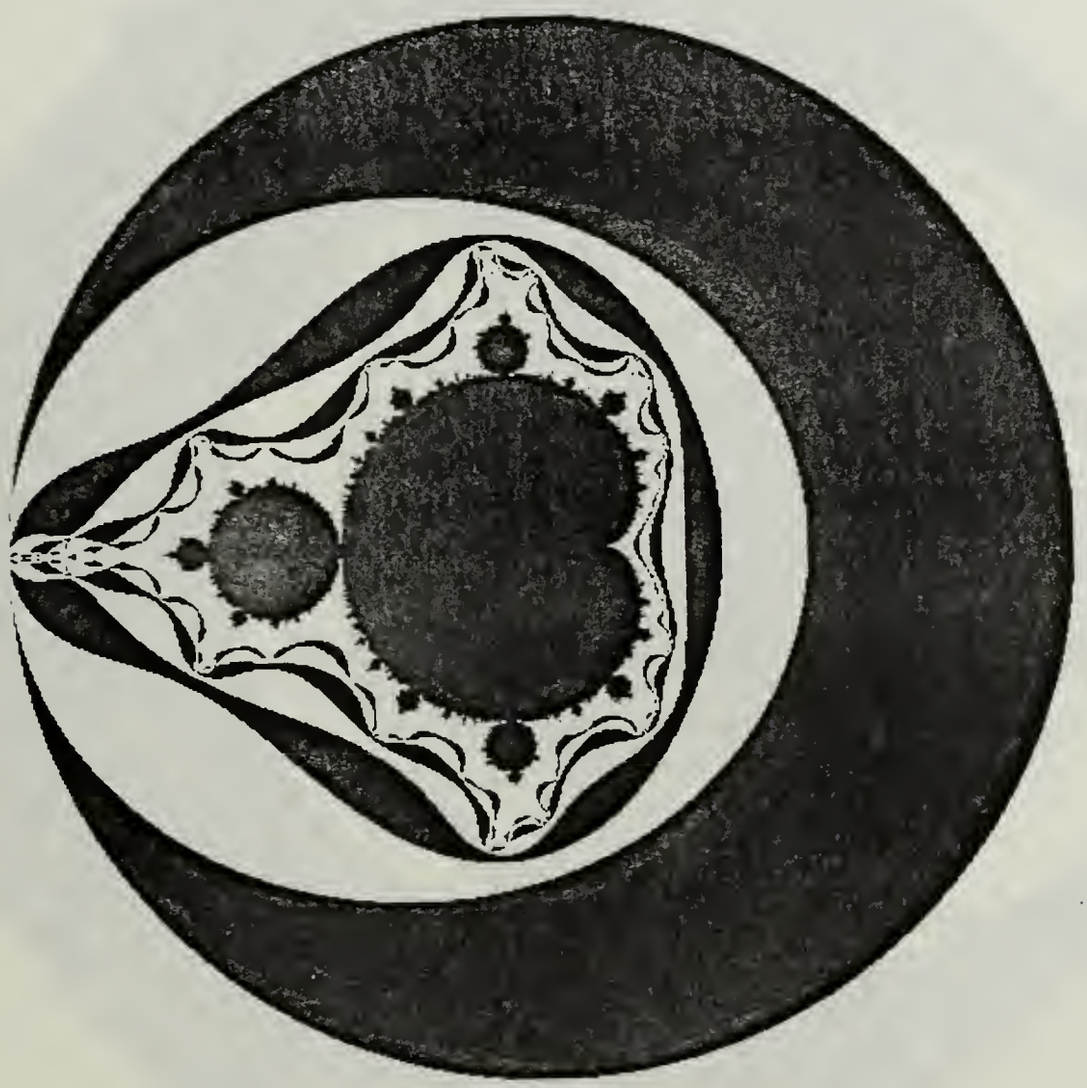


Figure 6.18 The Mandelbrot Set (8 Rings)

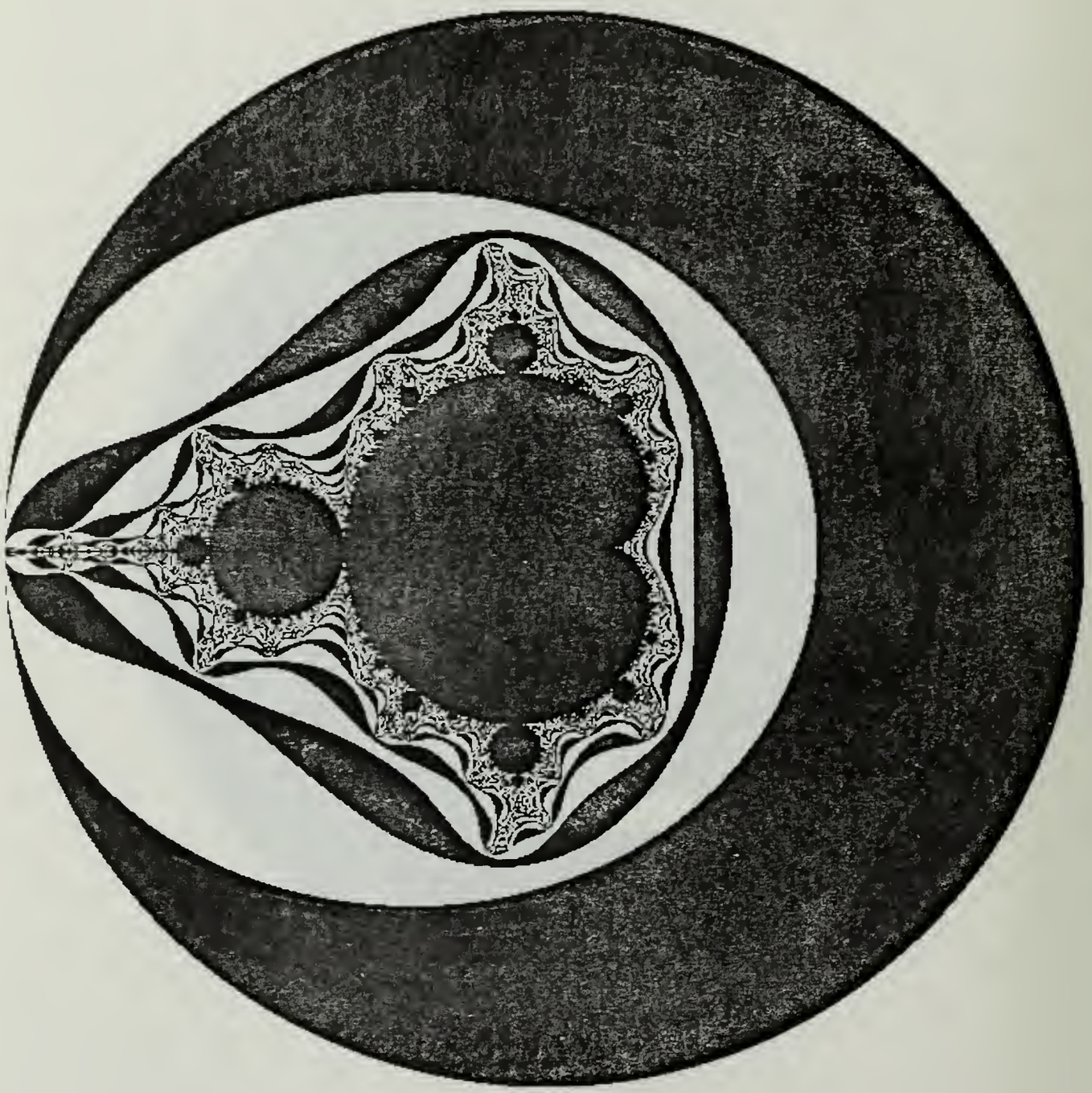


Figure 6.19 The Mandelbrot Set (50 Rings)

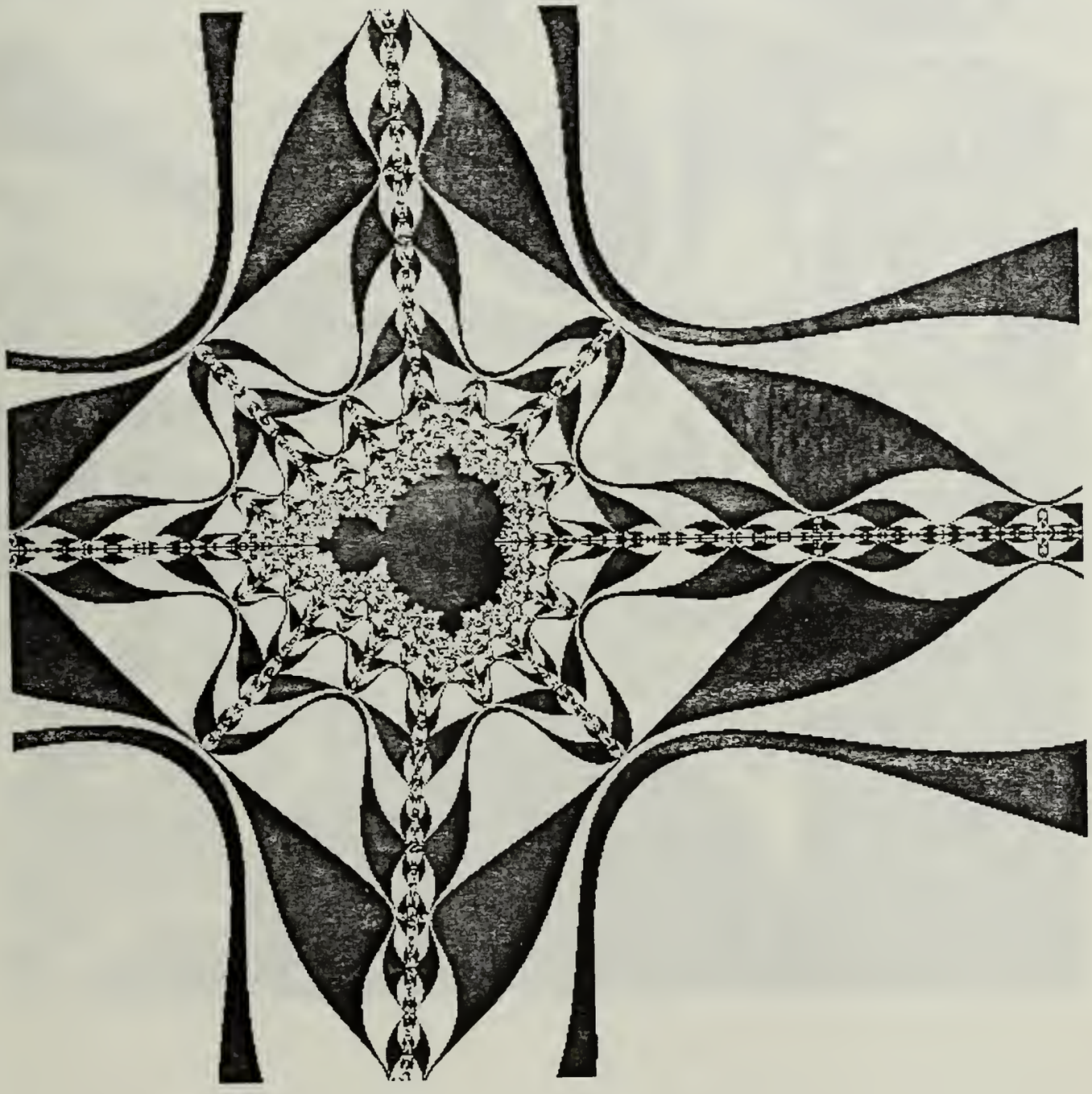


Figure 6.20 Antenna of the Mandelbrot Set

Saving the best for last, Figure 6.21 is the most aesthetically pleasing picture we have produced. Again every other color up to 50 iterations is displayed, and colors for 50 to 100 iterations are left white. The color scheme used on the Iris graphics work station is designed to provide gradual changes in color for five bands of iterations, with maximum color contrast at the border of the bands. The portion of the complex plane displayed is from -1.781 to -1.768 along the real axis and from 0.0 to +0.13 along the imaginary axis. This figure is a recreated version of map 33 in [Ref. 5: p. 80].

The figures presented here are not an all inclusive set of figures that can be produced by the two programs. There is an almost unlimited number of pictures that can be constructed. We present here only a few illustrative examples.

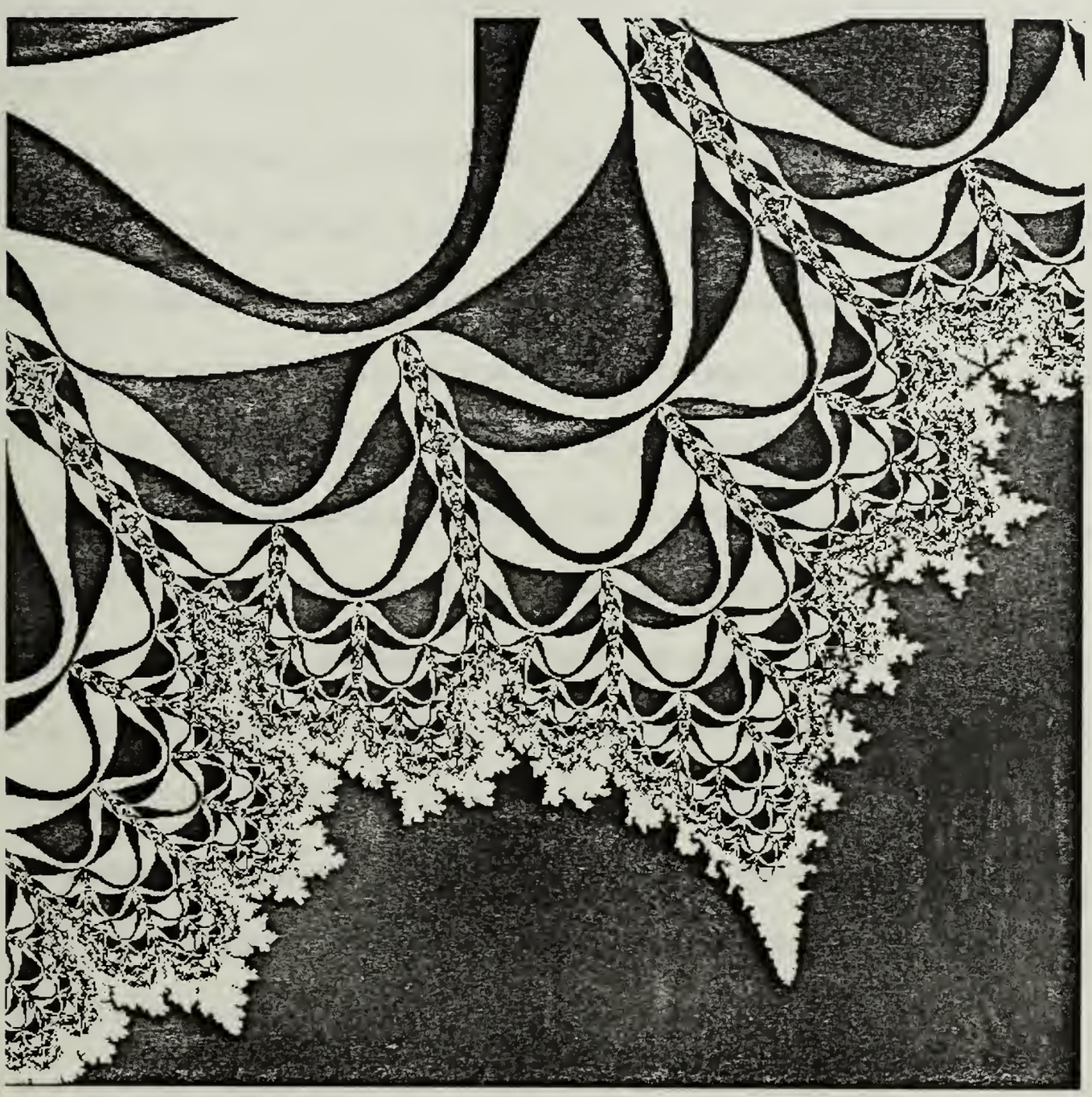


Figure 6.21 [Ref. 5: p. 80] Map 33 Rendering

# VII. CONCLUSIONS

## A. SUMMARY

We have implemented an interactive workshop for constructing and displaying two-dimensional Koch-like fractal curves. This workshop facilitates learning about Koch-like fractals through experimentation. The workshop gives one the ability to rapidly construct a generator. select an initial object and then immediately display the resulting fractal curve. This immediate visual feedback allows one to gain an intuitive feel for the Koch-like fractal curves.

We have also implemented a program that allows visual exploration of the Mandelbrot set. One can specify the portion of the complex plane that is to be displayed. The Mandelbrot set can be viewed with different levels of magnification. The Mandelbrot set in Figure 6.18 displays approximately $\mathbf{16\ units^2}$ of the complex plane. The Mandelbrot set in Figure 6.21 displays only $\mathbf{2.89 \times 10^{-4}\ units^2}$, a magnification of approximately $\mathbf{5 \times 10^4}$.

## B. LIMITATIONS

The biggest limitation in the algorithm for Koch-like curves is that the algorithm only works with straight line segments. This restricts the program drastically. The ability to manipulate arcs and curves would greatly enhance the program.

Although not a severe limitation. the data structures limit the size of the generator to 21 line segments and the size of the initial objects to 25 line segments. These are arbitrary limits and can be easily changed. The algorithm for replacing initiators with generators is a recursive algorithm, and requires a terminating condition. We use a distance of one pixel's width for our terminating condition. If the length of a line segment is less than one pixel width, we terminate the recursion. Implicit in the algorithm is the length of the generator line segments relative to the initiator length. The generator line segments must be smaller than the initiator they are replacing. If all of the generator line segments are smaller than the line segment the generator is replacing, the terminating condition is eventually reached. If a generator line segment is longer than the initiator it is replacing, the program gets into an infinite loop condition and does not terminate naturally. In this loop condition, the program usually terminates when some system parameter is exceeded. The system condition that causes termination is most likely running out of stack space to save the recursive calls. The program then *crashes*. We have not prohibited this condition, although we display a warning as the generator is being constructed if a generator line segment is longer than the original line segment. Although not a normal mode of operation, if a generator has a line segment long enough to cause a crash, the displays prior to the system crash are sometimes interesting.

## C. AREAS OF FURTHER RESEARCH

As mentioned above, an algorithm that can create Koch-like curves using arcs and curves instead of straight line segments would aid in the ability to interactively experiment with fractals. The algorithm for displaying the Mandelbrot set could be expanded to allow the display of the Julia sets as well.

Fractals are still not well defined. Fractals have attracted a wide audience. and different forms of fractals are being used for a variety of tasks. Among these tasks is the generation of music, the rendering of plants and mountains, the display of the Julia set "monsters", and the study of the distribution of natural rain [Ref. 3: pp. 1-45]. All of these tasks use fractals, although the fractals one task uses does not necessarily resemble the fractals another uses. A better definition of fractals is needed. Perhaps what is needed is to define a partition of the set of fractals. Each class of the partition exhibiting special characteristics of that class. Further study is needed to determine if a partition of the set of fractals is feasible. If a partition can be made, then new names can be given to the classes of the partition, and some of the confusion that exists with the term fractals could be eliminated.

## D. CONCLUSIONS

It is the hope of the author that this work, and the two-dimensional fractal workshop can be used to learn about fractals. The concepts presented are not too complex, but can be confusing at times. The pictures that can be displayed are

worth the time and effort of working with the workshop. Running the workshop requires no knowledge of the algorithm that constructs the curves. However, a little understanding of the algorithm can help give the curves more meaning and help develop an intuitive understanding of fractals.

# APPENDIX A – THE MANDELBROT SET

This section contains the computer listing of the function used to calculate the Mandelbrot set. The function saving data for the laser printer is not included.

```
#include "gl.h"
#include "device.h"
#define four 4.0

typedef struct {
      double  real;
      double imaginary;
      } complex;

main()
{

  /* local variables ********************************************/
   complex  Start;            /* the lower left corner of the area of interest*/
   complex  Z;                /* the complex number being manipulated       */
   complex  Zsquared;         /* an intermediate calculation  Z * Z         */
   complex  C;           /* the constant complex number in Z = Z * Z + C  */
   double   DeltaX;      /* the change per unit along the real axis       */
   double   DeltaY;      /* the change per unit along the imaginary axis  */
   double   Size;             /* the modulus of the complex number          */
   double   Scalefactor; /* the magnification factor for the picture      */
   int      iteration;        /* count of number of iterations*/
   int      maxrows=767;      /* maximum number for rows in the pixel space*/
   int      row;              /* the scan line being worked on currently*/
   int      maxcolumns=1023;/* max number of columns in the pixel space*/
   int      column;           /* the column of pixels now being worked on*/
   int      maximum=200;  /* the number of iterations that signal a cut off */
   char     color_table[4096];     /* for use with saving to printer*/
   Colorindex wmask;

/* Functions used by the main routine */

   complex  ComplexAdd();
   complex  ComplexMultiply();
   double   SizeofSquared();/* for determining the modulus */
```

```
void    setupcolors();          /* for the iris */
void    dumpbits();         /* save for laser printer */
void    colortable();       /* for the laser printer*/
void    getthe();               /* gets a filename*/


/* initialize the graphics package */
ginit():
doublebuffer():
gconfig();
wmask=((1<<getplanes()) -1);
writemask(wmask);

/*  select full screen viewport  */
viewport(0.1023.0,767);

/*  orthogonal projection 2D for the world coordinates sys  */
ortho2(0.0,1023.0.0.0.767.0):

/* clear both buffers */
color(BLACK);
clear();
swapbuffers();
clear();

/* turn the cursor off*/
cursoff();


while(TRUE)
{
  /*************************************************/
  /* input the starting left corner of the area of interest  */
  /*************************************************/

    /* put a box around the place to put the message */
    color(BLUE);
    rectf(600.0,700.0,1000.0,750.0);

    /* move to the place to print */
    cmov2(605.0,730.0);
```

```
/* print the message */
color(WHITE):
charstr(" Input the lower left corner of interest");
cmov2(605.0.710.0);
charstr("      +/-)n.nn (+/-)n.nn");

/* put that info on the screen */
swapbuffers():

/* now get the info from the keyboard  */
scanf("%f%f",&Start.real.&Start.imaginary);

/*************************/
/* now get the sizeing factor */
/*************************/

/* put a box around the place to put the message */
color(BLUE):
rectf(600.0.700.0.1000.0.750.0):

/* move to the place to print */
cmov2(605.0,730.0);

/* print the message */
color(WHITE);
charstr("Input the sizing factor");
cmov2(605.0,710.0);
charstr(" >1 for more area <1 for magnification");

/* put that info on the screen */
swapbuffers();

/* now get the info from the keyboard  */
scanf("%f",&Scalefactor);

/* erase the message */
color(BLACK);
rectf(600.0,700.0,1000.0,750.0);

/* put in a thank you message */
color(WHITE);
cmov2(605.0,710.0);
charstr(" Thank you for the information ");
```

```
    /* put that info on the screen */
    swapbuffers();

    /* set up the color ramp */
    setupcolors();

/************************************/
/* now set up for the fractals calculation */
/************************************/

    /* figure out the unit increment values */
    DeltaX = Scalefactor / maxcolumns;
    DeltaY = (maxrows / maxcolumns ) * Scalefactor / maxrows;

    /* note the size of the screen for the picture now is 767 x 1023 */
    /*  maxrows / maxcolumns keeps the aspect ratio for the window */

    /* write to both buffers so you can see it as you go */
     frontbuffer(TRUE);
     backbuffer(TRUE);

    /* for each pixel in the display area, by row look at each pixel */

    for (row = 0; row < maxrows; row = row + 1)
    {
    for (column = 0; column < maxcolumns; column = column + 1)
     {
       /* set Z to zero initially */
       Z.real = 0.0;
       Z.imaginary = 0.0;

       /* initialize C for the new pixel */
       C.real = column * DeltaX + Start.real;
       C.imaginary = row * DeltaY + Start.imaginary;

       /* reset the iteration counter */
       iteration = 1;

       /* add Z and C */
       Z = ComplexAdd(Z,C);

       /* see how large the modulus has gotten */
       Size = SizeofSquared(Z);
```

```
/* if the iterations have exceeded maximum that chances are
   the number will always stay in the set.
   And if the size is > 4 it will eventually go to infinity
   so it is not in the set  */

while ((iteration < maximum) && ( Size < four))
 {
    /* find Z*Z  */
    Zsquared = ComplexMultiply(Z,Z):

    /* the heart of the algorithm  Z = Z*Z +C */
    /* Add Zsquared + C */
    Z = ComplexAdd(Zsquared,C);

    /* again find how big the number is getting */
    Size = SizeofSquared(Z):

    /* increment the iteration counter */
    iteration = iteration + 1:

  } /* end while iteration < max and size less than 4 */

  /*****************************************************/
/* now that you are out of the loop find out what to do next */
  /*****************************************************/

/* is the size still less than four? */
if (Size < four)
  {
  /* the size is still small so the iterations must have
     reached the limit, so the point is in the Mandelbrot
     set.  I choose to leave the color black for this */
  color(BLACK);
     }

  if (Size >= four)
  {
  /* use the number of iterations as an index into the
     color map for the color this pixel is to be printed */
     color(iteration);
     }
```

```
        /* now move the graphics position to the pixel desired */
        move2((double)column,(double)row);

        /* and print the color desired */
        pnt2((double)column,(double)row);

      } /* end for column */
    }  /* end for row    */

/*************************************************/
/*  this is the part that saves the info for the printer */
/*************************************************/

  /* save the state of the program */
  pushattributes():
  pushmatrix():
  pushviewport();

  getthe(filename):

  /* set the new viewport and ortho2 */
  viewport(0,1023,0,767);
  ortho2(0.0,1023.0,0.0,767.0);

  /* which colors do you want printed? */
  colortable(color_table);

  /* dumpbits reads the back buffer so put what you want there */
  swapbuffers();

  dumpbits(0.0,0.0,600.0,600.0,color_table,"filename");

  /* restore the state of the program */
  swapbuffers();
  popviewport();
  popmatrix();
  popattributes();

  /* this says only do it once, remove to do more */
  break;

  } /* end while true */
} /* end main  */
```

```
/************************************/
/*  Routines to do the complex arithmetic */
/************************************/

complex ComplexAdd(Number1,Number2) /*  Add two complex numbers */

  complex Number1:  /* the first number to be added */
  complex Number2:  /* the second number to be added */
   {
    /* local variable to hold the answer */
    complex Answer;

     Answer.real = Number1.real + Number2.real;
     Answer.imaginary = Number1.imaginary + Number2.imaginary;
     return(Answer):
    }

complex  ComplexMultiply(Number1,Number2) /*  Multiply two complex numbers */

  complex Number1:  /* the first number to multiply */
  complex Number2:  /* the second number to multiply */
  {
    /* local variables */
    complex  Answer:

    Answer.real = (Number1.real * Number2.real)-
                  (Number1.imaginary * Number2.imaginary);
    Answer.imaginary = (Number1.real * Number2.imaginary) +
                (Number1.imaginary * Number2.real);
    return(Answer);
   }

 double   SizeofSquared(Number) /* find the modulus squared of a comples number */

     complex   Number;    /* the input number */
   {
     /* local variables  */
     double   Answer;  /* the value returned */

     Answer = Number.real * Number.real + Number.imaginary * Number.imaginary;
     return(Answer);
   }
```

## APPENDIX B – CALCULATING GENERATOR POINTS

```
/*********************************************************
PURPOSE   :   this is the function that takes the initiator and
              replaces it with the generator, does so recursively
PARAMETERS :  X1,X2. Y1,Y2  the end points of the initiator
*********************************************************/


#include "Init.extern.h"  /* Global variables include file; external
                          variables maintained in the main routine
                          Koch.c                              */
#include <stdio.h>
#include <math.h>         /* Standard math include file for UNIX lib */

  /* BEGIN RECURSIVE PROCESS */

     generate(X1,Y1,X2,Y2)

          /* Parameter variables */
            double X1,Y1,X2,Y2;

     {

     /* Declare variables; Recursive parameter and local variables
       are passed "by value" so for each call the current variables
       are preserved by the system as any normal subroutine call would
       (on a stack). The values are restored as the recursive routine
       "backtracks" after hitting the recursive termination event
       (distance between points < pixel width )                  */

          /* Local variables */

          double  Temp1;              /* used in tangent calculations*/
          double  G_point[20][2];     /* array to hold the generator points*/
          double  Slope_init,Slope_perp,Slope_gen,Slope_end_gen;
          double  X_perp,Y_perp;      /* where perp line crosses initiator*/
          double  TEMP,DIST;          /* to find dist between two points*/
          double  X_unk,Y_unk;        /* the point we are trying to find*/
          double  b_perp,b_gen,b_end_gen;   /* the y intercepts */
          int     I,J;                /* loop control variables*/
```

```
    /*  keep track of the level of recursion on the way in */
Recursion_depth = Recursion_depth + 1;/* global variable*/

    /* The Koch curve is defined in the infinite but our recursion
    will terminate after the distance between points becomes less
    than the length of a pixel.  The window is declared
    so there is a one to one correspondence between the
    Euclidian space (natural numbers of) and the pixels
    of the screen.                                    */

    /* Determine distance between point 1 and point 2 */
TEMP = (X2 - X1)*(X2 - X1) + (Y2 - Y1)*(Y2 - Y1):
DIST = sqrt( TEMP );

    /* IF DIST less than one then terminate this recursion and   */
    /* save the point for the printer file and begin backtracking */

if (DIST <=  1.0)
{
    return;
  } /* END  if DIST < 1   */

/* ELSE DIST > 1, Construct the GENERATOR on top of the INITIATOR  */

    /* Put INITIATOR points one and two into the first and last */
    /* points of the GENERATOR points array as they are always  */
    /* part of the generated structure                          */

G_point[1][0] = X1;
G_point[1][1] = Y1;
G_point[init.Generator_points + 2][0] = X2;
G_point[init.Generator_points + 2][1] = Y2;

    /* Determine the slope of the line formed by the initiator end points*/
    /* This is the slope of the INITIATOR                                */

if (X2 == X1)
{
    Slope_init = 10,000;
  }
```

```
else
{
    Slope_init = (Y2-Y1)/(X2-X1);
  }

    /* For each GENERATOR point (except end points as they are equal*/
    /* to the INITIATOR end points) find the X,Y values. This is    */
    /* accomplished by using the data in the data dictionary include*/
    /* file Init.h (Init.extern.h) and the fact that the ratios     */
    /* and angles between the INITIATOR and GENERATOR points remain */
    /* the same regardless of the INITIATORS length or orientation  */
    /* in EUCLIDIAN space                                           */

for (I=1; I <= init.Generator_points; I++)
{
    if (init.Gen_ratio[I] == 0.0)
    {  /* point is perpendicular at X1,Y1 */

        if (init.Y_ratio[I] == 0.0)
        {  /* the point is the start point */

                G_point[I+1][0] = X1;
                G_point[I+1][1] = Y1;
          }

        else
        {    /* calculate slope_init */

            if (X1 == X2)  /* initiator is vertical */
            {
                Slope_init = 10000.0;
              }
            else
            {
                Slope_init = (Y2 - Y1)/ (X2-X1);
              }
          }  /* end else calculate slope_init */

            /* calculate slope of generator line  : note the    */
            /* generator line is perpendicular to the initiator */
```

```
if ((Slope_init == 0.0) || (Slope_init >= 10000.0))
{
     if (Slope_init == 0.0)
     {
          Slope_gen = 10000.0;
       }
     else
     {
          Slope_gen = 0.0;
       }
  } /* end slope_init = 0 or 10000 */

else  /* initiator not parallel to an axis */
{
     Slope_gen = (-1.0)/ Slope_init:
  }

     /* find the y intercept of generator line */
b_gen = Y1 - (Slope_gen * X1):

     /* now have an equation for the generator line    */
     /* Y_unk = Slope_gen*X_unk + b_gen                */
     /* two unknowns                                   */

     /* find the slope of the line from the generator  */
     /* point to the end point                         */
Slope_end_gen = (Slope_init - init.Y_ratio[I])/
                (1.0 + (Slope_init * init.Y_ratio[I]));

     /* find the y intercept for the end gen line */
b_end_gen = Y2 - (Slope_end_gen * X2);

     /* now have a second equation for a line through*/
     /* the unknown gen point                          */
     /* Y_unk = Slope_end_gen * X_unk + b_end_gen      */
     /* two eqns, two unknown, solve for X_unk         */
X_unk = (b_end_gen - b_gen)/(Slope_gen - Slope_end_gen);

     /* substitute in and find Y_unk */
Y_unk = Slope_end_gen * X_unk + b_end_gen;

     /* put that point in the array*/
G_point[I+1][0] = X_unk;
```

```
        G_point[I+1][1] = Y_unk;

    } /* end init.Genratio = 0 */
else   /**  ratio is not 0 **/
{
        /* Using the ratios of the generator perpendicular*/
        /* intercept points on the INITIATOR determine the*/
        /* X.Y values of the point of intersection of the*/
        /* perpendicular from the GENERATOR point to the */
        /* INITIATOR line.                                */

        X_perp = X1 + init.Gen_ratio[I] *(X2 - X1);
        Y_perp = Y1 + init.Gen_ratio[I] *(Y2 - Y1):

/* If the angle of the INITIATOR point 1 and the GENERATOR */
        /* point in question is zero then the GENERATOR point is   */
        /* coincident with the INITIATOR line and no further       */
        /* calculations are necessary                              */

        if (init.Gen_angle[I] == (double)0.0 )
        {
            G_point[I+1][0] = X_perp;
            G_point[I+1][1] = Y_perp;
          }
        else  /* gen angle not equal to 0 */
        {
                /* There are three STATES possible at this time.
                 * STATE 1
                 * where the slope of the initiator line is parallel
                 * to the X or Y axis (which causes havoc with the line
                 * equations).
                 * STATE 2 where the slope of the line formed
                 * by the initiator point 1 and the unkwown generatorpoint
                 * is parallel to the X or Y axis.
                 * Or STATE 3 where no lines are parallel to any axis.
                 * Because of the relation of the initiator line and
                 * generator line both cannot be parallel simultaneously. */

         /* Determine the slope of the line through the INITIATOR */
                /* start point and the unknown GENERATOR point using the */
                /* tangent of the Gen_angle in Init.h                    */
                /* tan (A + B) = (TanA * Tan B) / (1 - TanA * TanB )     */
```

```
        Temp1 = init.Tan_theda[I] * Slope_init;

    /* check to avoid / by zero */
    if (Temp1  == 1.0)
    {
        Slope_gen = 10000.0:
      }
    else
    {
      Slope_gen=(init.Tan_theda[I] + Slope_init)/
            ((double)1.0-init.Tan_theda[I] * Slope_init);
      }
    if ((Slope_gen != (double) 0.0) &&
            (Slope_gen < (double) 10000.0))
    {

            /* Condition one of STATE 3 */
            /  generator not parallel to any axis */

/* Determine Y-intercept for the generator line */
/*  y = mx +b .....b = y- mx */
       b_gen = Y1 - (Slope_gen * X1);

       if ((Slope_init == (double) 0.0) ||
            (Slope_init >= (double) 10000.0))
       {
     /* STATE 1 - INITIATOR parllel to an axis   */
     /* find out which axis                      */

                if (Slope_init >= (double) 10000.0)
            {
                 /* STATE 1 condition 1; INITIATOR is parallel */
                 /* to the Y axis  Y part will be Y_perp       */
                 /* G_point[I+1][1] = Y_perp;                  */
                 /* x = (y-b)/m                                */

            G_point[I+1][0] = (G_point[I+1][1] -
                    b_gen)/Slope_gen;

              }
            else /* slope_init = 0.0, parallel to x axis  */
            {
```

```
            /* STATE 1 condition 2; INITIATOR is parallel
              to the X axis X part will be X_perp */
                G_point[I+1][0] = X_perp;

                    /* y = mx +b   */
                G_point[I+1][1] = Slope_gen *
                      G_point[I+1][0] +b_gen:
                       }
            } /* END STATE 1 */
    else
    { /* neither initiator or generator slope = 0.0 or 100000.0 */
          /* STATE 3 */

          /* Determine slope of perpendicular line through the*/
          /* INITIATOR perp. intercept.    mperp = -1/m   */
             Slope_perp = ( -1.0)/Slope_init:

          /* Determine Y-intercept for perpendicular line */
          /* b = y - mx   */
             b_perp = Y_perp - (Slope_perp * X_perp);

           /* Determine the X.Y values of the unknown */
           /* GENERATOR point. First solve for X */

             G_point[I+1][0]=(b_perp -b_gen)/
                 (Slope_gen - Slope_perp);

           /* now X is known  y = mX + b          */
        G_point[I+1][1] = Slope_gen *
                     G_point[I+1][0] + b_gen;
      }
} /* END STATE 3 cond. 1 if */
else  /* line to generator point is parallel to an axis  */
  {
      /* STATE 2 */ /* mperp = -1/ m  */
    Slope_perp = ( -1.0)/Slope_init;

                  /* b = y - mx */
    b_perp = Y_perp - (Slope_perp * X_perp);

    if (Slope_gen >=  10000.0)
    {               /* parallel to Y axis   */
```

```
                G_point[I+1][0] = X1;
                                /* y = mx + b */
                G_point[I+1][1] = Slope_perp *
                                G_point[I+1][0] + b_perp;
              }
            else        /* parallel to the X axis  */
            {
                G_point[I+1][1] = Y1:
                            /* x = (y - b) / m  */
                G_point[I+1][0] = (G_point[I+1][1]- b_perp)/
                                Slope_perp;
              }
          }     /* end else gen parallel to an axis */
      } /* END IF GEN_ANGLE = 0 */
    } /* END ELSE x ratio is not 0 */
} /* END FOR */

      /* Poll the screen */
    Menu_gen();

      /* Draw GENERATOR  */
    Draw_gen(G_point);

      /* Start recursion on each line formed by the generator */

    for (J=1; J <= init.Generator_points + 1 ; J++)
    {
      generate(G_point[J][0] ,G_point[J][1],
                  G_point[J+1][0],G_point[J+1][1]):

      /* going back to caller reduce recursive level by 1 */
      Recursion_depth = Recursion_depth - 1;

    }
}     /* END generate */
```

## LIST OF REFERENCES


1. Mandelbrot, Benoit B., *The Fractal Geometry of Nature,* W. H. Freemond Company, 1983.

2. Special Interest Group on Computer Graphics of the Association for Computing Machinery (SIGGRAPH), *Fractals: Basic Concepts Computation and Selected Topics,* SIGGRAPH 86 Course No. 11 Notes, August 18, 1986.

3. Sorensen, Peter R., "Fractals," *Byte,* v. 9, no. 10, pp. 157-172, September 1984.

4. Gaddis, Michael E., *The Fractal Geometry of Nature: Its Mathematical Basis and Application to Computer Graphics,* Master's Thesis, Naval Postgraduate School, Monterey, California, June 1986.

5. Pietgen, Heinz-Otto and Richter, Peter H., *The Beauty of Fractals,* Springer-Verlag, 1986.

6. Dewdney, A. K., "Computer Recreations," *Scientific American,* v. 253, pp. 16-24, August 1985.

7. Byrnes, Edward A., "Basic Fractals," *Sextant,* v. 23, pp. 76-87, July-August 1986.

# INITIAL DISTRIBUTION LIST

| | | No. Copies |
|---|---|---|
| 1. | Defense Technical Information Center<br>Cameron Station<br>Alexandria, Virginia 22304-6145 | 2 |
| 2. | Library, Code 0142<br>Naval Postgraduate School<br>Monterey, California 93943-5002 | 2 |
| 3. | Chairman, Code 52<br>Department of Computer Science<br>Naval Postgraduate School<br>Monterey, California 93943-5000 | 1 |
| 4. | Computer Technology Curricular Officer, Code 37<br>Naval Postgraduate School<br>Monterey, California 93943-5000 | 1 |
| 5. | Cdr Patrick Moneymaker<br>Commanding Officer<br>Strike Fighter Squadron One Nine Five (VFA 195)<br>FPO San Francisco, California 96601-6237 | 1 |
| 6. | Dr. Robert L. Mason<br>1676 Shorecrest Road<br>Moses Lake, Washington 98837-0000 | 1 |
| 7. | Dr. Michael J. Zyda, Code 52Zk<br>Department of Computer Science<br>Naval Postgraduate School<br>Monterey, California 93943-5000 | 2 |

| | | |
|---|---|---|
| 8. | Cdr Ronald E. Rautenberg, Code 52Ra<br>Department of Computer Science<br>Naval Postgraduate School<br>Monterey, California 93943-5000 | 1 |
| 9. | Cdr Lewis G. Mason, Code 52Ma<br>Department of Computer Science<br>Naval Postgraduate School<br>Monterey, California 93943-5000 | 6 |